National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

University of Alberta

A RISC DESIGN INCLUDING INCREASED I/O BANDWIDTH AND BACKGROUND REGISTER SAVES

by

Norman Jantz

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Electrical Engineering

Edmonton, Alberta
Spring, 1990

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

ISBN 0-315-60182-5

NL-339 (r. 88/04) c

Canada

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Norman Jantz

TITLE OF THESIS: A RISC Design Including Increased I/O Bandwidth and Background Register Saves

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1990

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) ......Norman Jantz......
Permanent Address:
11523 - 76 Avenue
Edmonton, Alberta
Canada

Dated: December 15, 1989

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **A RISC Design Including Increased I/O Bandwidth and Background Register Saves** submitted by **Norman Jantz** in partial fulfillment of the requirements for the degree of **Master of Science.**

Emil Girczyc
Supervisor: Dr. Emil Girczyc

Keith A Stromsmoe
Dr. Keith Stromsmoe, Dept. of E.E.

Dr. Ahmed Kamal, Dept. of C.S.

Date Dec 15, 1989

Abstract

This thesis presents a novel 32 bit RISC architecture which achieves greater I/O bandwidth and executes fewer nonprocessing states than conventional RISC designs. Greater I/O bandwidth is achieved by encoding 2 instructions per 32 bit word, moving the program counter offchip, and by utilizing unidirectional busses. Fewer nonprocessing states are achieved by "background" register saves and restores and with pipelined memory accesses and instruction execution. The architectural design of the processor combines these novel features with good features of previous RISC designs to meet philosophical goals such as efficient use of I/O bandwidth and locality of information processing. The implications of the unconventional division between processor and external memory controller on the system level architecture are then discussed. Next, circuit designs and floorplans of the implemented IC are shown to verify that this architecture could be fabricated in a conservative CMOS technology. Finally, a summary of emulation and testing results is presented.

Acknowledgements

I would like to thank my supervisor, Dr. Emil Girczyc, for his guidance and support throughout this research project. As well, I would like to thank Dr. Jack Mowchenko and other faculty members within Computer Engineering who showed patience and understanding as well as admirable professional qualities. In the design of ARC, several fellow students contributed in the initial stages including Duncan Glendenning, Tai Ly, Trevor May, and Mike Smith. In this research endeavor, many people contributed helpful suggestions to problems which had to be overcome. In this respect, I would like to thank Mak Paranjape, Stuart Olsen, Harold Peacock, Steve Sutphen, Rene Leiva, Trevor May, Trevor Monson, Doug Konrad, Ash Parameswaren, Joe DeAlmeida, and Scott Stephens. Most importantly, however, I would like to thank my parents and sisters, who were not only encouraging and supportive throughout this project, but also showed a great deal of understanding during my university education.

Table of Contents

1. Introduction 1
2. Previous Research 5
    2.1 Instruction Set 6
    2.2 Load and Store Architecture 7
    2.3 Registers 8
        2.3.1 Special Purpose Registers 8
        2.3.2 General Purpose Registers 8
    2.4 Pipelining 9
    2.5 Caching 12
        2.5.1 Caching Techniques 12
        2.5.2 Overview of Instruction and Data Caching 13
    2.6 Architectural Features of Stanford's MIPS 14
    2.7 Architectural Features of the Berkeley RISC II 15
    2.8 Summary of Previous Research 18
3. Novel Concepts That Increase Speed and Efficiency 20
    3.1 Two Instructions per 32-bit Word 21
    3.2 Unidirectional I/O 22
    3.3 Background Register Saves and Restores 23
    3.4 Pipelined Memory Access and Instruction Execution 24
    3.5 External Program Counter 26
    3.6 Intelligent Memory 27
    3.7 Philosophy Behind ARC and MCU Design 28
4. Architecture 31
    4.1 ARC Processor Architecture 31
        4.1.1 Datapath 31
            4.1.1.1 The ALU 32
            4.1.1.2 The Register File 32
        4.1.2 The I/O and Control Unit 33
        4.1.3 Instruction Set 34
            4.1.3.1 ALU Instructions 34
            4.1.3.2 Transport Instructions 36
            4.1.3.3 Control Instructions 45
        4.1.4 Pipeline and Resource Scheduling

4.2 ARC System Architecture 56
4.2.1 Interface of ARC to Memory Control Unit 57
4.2.1.1 PC and Special Registers of the MCU 58
4.2.1.2 The Register Stack of the MCU 59
4.2.1.3 MCU I/O Control 59
4.2.1.4 MCU Caching 60
4.2.1.4 Optional MCU Functions to Enhance Performance
5. ARC Circuit/Hardware Design Description 66
5.1 The Arithmetic and Logic Unit 66
5.1.1 ALU Adder 68
5.1.2 ALU Decoder 68
5.1.3 ALU Bitslice Circuitry 68
5.1.3.1 ALU Bitslice Schematics 71
5.1.4 Condition Tester 74
5.1.4 Condition Tester 74
5.1.5 Other Non-Bitslice ALU Circuitry 77
5.2 Register File 78
5.2.1 Register Array 82
5.2.2 Subwindow Decoder 84
5.2.3 Subwindow Enables 86
5.3 The I/O and Control Unit (IOCU) 88
5.3.1 I/O Bitslice Circuitry 90
5.3.2 Instruction Extraction and latching Circuitry 92
5.3.3 Transport and Control Instruction Circuitry 94
5.3.4 Window Control Logic 97
6. Analysis of Concepts 107
6.1 Emulation of Architecture 107
6.2 Analysis of Architectural Features 107
6.2.1 Bandwidth Utilization/Availability 109
6.2.2 Bandwidth Balancing 111
6.2.3 Bandwidth Available for Background Register Saves/Restores 112
6.2.4 Nonprocessing States 115
6.2.5 Two Instructions Per Word 115
6.2.6 External Program Counter 116
6.2.6 Unidirectional IN and OUT Busses 117
6.2.6 Testing Results 117
7. Conclusions 119
7.1 Architectural Concepts 119
7.1.1 2 Instructions per Word 119
7.1.2 External Program Counter 120
7.1.3 Unidirectional IN and OUT Busses 120
7.1.3 External Memory Control Unit 121

7.1.5 Background Register Saves/Restores 121
7.2 Further Work 121
7.2.1 Enhancements to Instruction Set 121
7.2.2 Improvements to Architecture/System 123
7.2.3 Improvements to Hardware 124
References 126
Appendix A: ARC Emulator Listings 128
Appendix B: ARC Implementation of Example Algorithms 154

List of Figures

Figure 2.1: Circular Buffer Organization of Overlapped Windows........................Page 10
Figure 4.1: Photograph of Bonded ARC Die..Page 32
Figure 4.2: ARC Block Diagram...Page 33
Figure 4.3: ALU Block Diagram...Page 34
Figure 4.4: Register Stack Structure...Page 37
Figure 4.5: I/O and Control Unit Block Diagram...Page 39
Figure 4.6: Instruction Formats...Page 43
Figure 4.7: Pipeline and Resource Scheduling...Page 49
Figure 4.8: Timing of Load Instruction..Page 51
Figure 4.9: Timing of Store Instruction ....Page 53
Figure 4.10: Timing of First-Halfword ALU Instruction ....Page 54
Figure 4.11: Timing of Second-Halfword ALU Instruction.......................................Page 55
Figure 4.12: ARC Interface to Single-Port RAM ....Page 58
Figure 4.13: ARC Interface to Dual-Port RAM...Page 59
Figure 4.14: ARC System Diagram...Page 60
Figure 5.1.1: ALU Floorplan ....Page 67
Figure 5.1.2: ALU Decoder PLA Equations ....Page 70
Figure 5.1.3: ALU Bitslice Floorplan...Page 72
Figure 5.1.4: ALU Bitslice Schematics ....Page 73
Figure 5.1.5: Non-Bitslice ALU Support Schematics...Page 75

Figure 5.1.6: ALU Transistor Schematics ....Page 76

Figure 5.1.7: Condition Tester Floorplan ....Page 78

Figure 5.2.1: Register File Floorplan........Page 80

Figure 5.2.2: Register-Pair Cell Floorplan ....Page 83

Figure 5.2.3: Register Bit Schematic........Page 83

Figure 5.2.4: Subwindow Decoder Floorplan ....Page 85

Figure 5.2.5: Example Decoder Schematic ....Page 86

Figure 5.2.6: PLA Equations for Enable Circuitry ....Page 86

Figure 5.3.1: I/O and Control Unit Floorplan........Page 89

Figure 5.3.2: Instruction Field Extraction/Multiplexing and Latching........Page 93

Figure 5.3.3: I/O Control Opcodes and Decoder PLA Equations........Page 95

Figure 5.3.4: I/O Control Operations Decoding and Latching........Page 96

Figure 5.3.5: Control Operations D_w Circuitry ....Page 98

Figure 5.3.6: 4-Bit Shift Register (SR4) Schematic ....Page 100

Figure 5.3.7: Current Window Pointer (CWP) Schematic ....Page 100

Figure 5.3.8: SWP Circuitry Design, Schematics, and Floorplan........Page 103

Figure 5.3.9: 3-Bit Counter Schematics ....Page 104

Figure 5.3.10: MMU Bus Window and Address Multiplexing........Page 106

List of Tables

Table 4.1: ARC Instruction Opcodes ....Page 42

Table 4.2: Instruction/Operation Restrictions ....Page 57

Table 6.1: Dynamic Benchmarks for Various Algorithms ....Page 108

Table 6.2: Static Instruction Statistics ....Page 110

Table 6.3: Dynamic Instruction Statistics ....Page 110

Table 6.4: Dynamic Push and Pop Opportunities ....Page 110

1. INTRODUCTION

This thesis presents a Reduced Instruction Set Computer (RISC) architecture which combines *standard* RISC elements and ideas from previous research with a number of novel ideas. The result is a novel 32 bit RISC architecture with a number of advantages over conventional RISC architectures.

The RISC concept was first presented by Patterson and Ditzel in 1980 [Patt80]. The concept of RISC was presented because of evidence showing the non-optimal utilization of silicon resources of conventional CISC processors due to the increased complexity of their instruction sets. [Kate 83] The *Reduced Instruction Set* concept initially implies a simplification and reduction in size of the instruction set. This results in a higher frequency of operation because instructions which are rarely used and would slow down the other instructions are not included in the RISC instruction set. Typically, this increase in operating frequency more than offsets the cost of emulating the omitted instructions resulting in greater system performance. Additionally, the RISC concept has come to imply efficient use of silicon resources and an addition of complexity only when it will lead to better system performance.

The Alberta Risc Chip (ARC) incorporates ideas from several RISC architectures which have been developed over the past several years. From this research several architectural features have emerged as *standard* RISC elements. These standard elements provide guidelines in terms of instruction set design, register architecture, compiler design, and pipelining.

In addition to the standard RISC elements, characteristics from Stanford's MIPS [Henn82,83] and the Berkeley RISC [Kate83] [Patt82] were incorporated into the ARC architecture. In particular, these characteristics include the following:

a) circular, overlapping register windows. [Kate83] [Patt82]

- b) non-interlocked pipeline, optimizing compiler [Henn82,83]
- c) delayed branches [Henn82] [Kate83]
- d) lack of condition code [Henn82]

These features have been combined with a number of novel architectural ideas resulting from a VLSI architecture course offered during the fall of 1986, including:

- a) 2 instructions per 32-bit word
- b) unidirectional I/O busses
- c) external PC; intelligent memory or Memory Control Unit(MCU)
- d) "background" register saves and restores to memory or MCU.
- e) pipelined access to memory

These features are combined in the ARC design to realize:

- a) high I/O bandwidth available
- b) reduced delays from bus turnaround and skew by utilizing unidirectional I/O busses
- c) fewer nonprocessing states
- d) versatile interface to memory and other subsystems.
- e) implementable in very conservative technologies (3 micron CMOS) and scalable to more aggressive technologies.
- f) low pin count

These are discussed in more detail in Chapter 3.

1.1. Conception of the Alberta Risc Chip (ARC)

This endeavor to design, implement, and test a RISC chip began as a VLSI architecture course taught by Dr. Emil Girczyc September through December, 1986. Five students enrolled in the course including Duncan Glendenning, Norman Jantz, Tai An Ly, Trevor May, and Mike Smith. All of those involved participated in design discussions. The objective of the course was to study current RISC architectures, design and implement a RISC processor incorporating some original ideas and improvements. Limitations included the available process technology(3 micron CMOS), IC design software, and human resources(5 Students, 4 months). Through the course of the term, the novel architectural concepts of ARC (presented on the previous page) evolved.

During the course, preliminary versions of an ALU(by Tai Ly) and a register file(by Norman Jantz) were implemented in 3 micron CMOS. Trevor May and Mike Smith did some work on an instruction unit, and Duncan Glendenning did a partial design of an on-chip memory management unit. Due to time limitations, the instruction and memory management units were not implemented.

1.2. Thesis Objectives

The objectives in preparing for this thesis have been:

a) to correct, improve, and complete the design outlined during the course.

b) to demonstrate that the architecture could be implemented in an available technology, (i.e.: 3 micron CMOS).

c) to verify the advantages of the architectural concepts

Architectural improvements to the class design include refining the instruction set, pipeline, intermodule communications, as well as resource and bus scheduling. In all cases the major architectural concepts remain the same while the implementation is defined or improved (see Chapter 4 and 5). The architecture was verified by an emulator program as described in Chapter 6.

Some ALU and register file hardware was designed during the course but architectural improvements and rigid area requirements led to a complete reimplementation of this circuitry. The instruction unit (IU) and on-chip memory management unit (MMU) were merged into the I/O Control Unit (IOCU) which was designed from scratch. The first full implementation of ARC was submitted for fabrication in Northern Telecom's 3 micron CMOS process during January, 1989. Due to implementation errors as described in Section 6.3, the design was only partially functional. Chapter 7 summarizes the evaluation of the novel architectural features and suggests future research.

2. PREVIOUS RESEARCH

In the past several years, there has been increasing interest in the RISC architecture philosophy. Many microprocessor manufacturers have subscribed to the RISC architecture philosophy and extensive research and development has been undertaken by both corporations and universities. From this research, several architectural features have become widely accepted as standard RISC elements [Stall88]. These standard RISC elements have been incorporated into ARC and are presented as follows:

- A limited and simple instruction set.
- A large number of registers.
- Load and store access to memory.
- All instruction operands in registers.
- A strategy of maximizing the use of registers and minimizing references to memory.
- The use of an optimizing compiler
- An emphasis on optimizing the pipelined execution of instructions to approach a throughput of 1 instruction per cycle.

A Summary of recent work in these areas will be presented below.

The two architectures most influential in the development of ARC, Stanford's MIPS [Henn82,83] and Berkeley's RISC [Kate83], will also be presented. The MIPS architecture was influential in terms of pipelining, instruction set, and philosophical issues. The Berkeley architecture was influential in terms of it's register file structure.

2.1. Instruction Set

In general, the size of the instruction set can be viewed as a tradeoff between hardware and software/compiler complexity. Increasing the size of the instruction set, for example, would increase the amount of ALU and instruction decoder logic, require more silicon area, and slow down the execution speed. Besides slowing down other instructions, a complex instruction or addressing mode is sometimes slower than a customized sequence of simpler instructions [Patt80], [Stal88]. The *Reduced Instruction Set* concept argues against large or complicated instruction sets for precisely the above reasons. On the software side, an increase in complexity is realized because the compiler has to deal with the following issues:

- emulation of complex instructions or addressing modes
- increased code size
- pipeline restrictions and operand dependencies
- code optimization and reordering
- reordering instructions after branches
- register allocation

For the compiler, it is a simple matter to emulate a complex instruction or addressing mode with a few simpler instructions. Because of this, the size of a RISC executable file is usually larger than a CISC executable. This results in a greater instruction bandwidth requirement.

Pipeline conflicts are a result of the way instructions utilize hardware resources. Having the compiler generate code that is free of pipeline conflicts simplifies the hardware because hardware resource allocation/interlocking is not required. To insure proper pipeline operation, the compiler must reorder instructions to insure a valid instruction sequence free of pipeline conflicts and satisfying all operand dependencies. The compiler must insure that an operand must be loaded into a register before being processed. This is achieved by ordering code such that a sufficient number of cycles separate the LoaD instruction of an operand and the first instruction processing this data. Most RISC compilers also optimize and reorder the code produced by packing independent instructions together, replacing NOPs, etc. Additional relationships between the pipeline and the compiler will be presented later in the pipelining discussion.

A register allocation algorithm is required by all RISC compilers. This algorithm ensures that the least number of LoaD instructions are used to load register operands into registers which reduces I/O bandwidth and increases performance.

2.2. Load and Store Architecture

In general, the number of addressing modes can be viewed as a tradeoff between hardware and software complexity. Several addressing modes, as in CISC architectures, would greatly increase the amount of instruction decoder and pipeline scheduling logic, require more silicon area, and slow down the execution speed. According to the RISC concept, only essential addressing modes should be used and addressing modes which are seldom used and which slow down the processor should not be included. Therefore, most RISC machines follow the load-and-store architecture.

Load-and-store machines have 2 types of instructions: a) instructions which transfer data between memory and registers, and b) instructions which process data in registers (ALU instructions). Therefore, this type of architecture usually only has 2 simple addressing modes for ALU instructions (Register-Register and Register-Immediate). The advantages of a load-and-store architecture include hardware simplicity and increased performance. The hardware simplicity results from the simplified execution and decoding of few addressing modes. In terms of added software complexity, compilers do not have a difficult time generating code for load-and-store architectures because they simply first get the operands, then use them. Performance is increased by keeping operands in registers so that processing does not stop for operands to be loaded from memory [Henn 82] [Kate 83].

2.3. Registers

Registers may be divided into special purpose and general purpose registers. Special purpose registers usually include control and status registers such as the program counter and the condition codes. General purpose registers are always user-visible and usually contain data and addresses. RISC processors differ greatly in the number and types of registers as described in the following subsections.

2.3.1. Special Purpose Registers

Each CPU architecture uses a different set of special purpose registers. While the MIPS machine [Henn82] has done away with a special condition code (CC) flag register, the current trend still seems to be towards including special purpose control and status registers [Stal88]. Most machines have a Program Status Word (PSW) which usually contains the Program Counter (PC), Flags (e.g.: Overflow, Carry, Negative, CC, interrupt), security codes, etc. Machines with PSWs can recover from interrupts by restoring the PSW and other special and general purpose registers.

2.3.2. General Purpose Registers

Increasing the size of the general purpose register file will increase performance because more temporal and spatial locality of an executing program is captured. As the size of the register file is increased, it becomes necessary for a subset of the registers to be "active" at a time because:

- It becomes difficult to set the optimum unstructured register utilization for a program with many procedures at compile time.
- Restrictions on number of bits available in an instruction for two(or more) register addresses.

- Overhead of context switches and procedure calls which must save all registers used as well as those storing global variables.

The register window concept was pioneered by the Berkeley RISC group who imposed a circular buffer structure on their register file [Patt82] [Kate83]. This circular buffer structure facilitates procedure calls and returns and parameter passing between procedures via overlapping register windows. Research done by members of the Berkeley RISC team indicates that about 94% of procedure activations are passed fewer than 5 arguments and use fewer than 12 words of arguments and local scalars. Therefore, an overlap of 4-8 registers and approximately 8-12 local registers per activation will usually be enough for a typical procedure activation. Fig. 2.1 (extracted from [Kate83] of the Berkeley RISC group) shows the "Circular Buffer Organization of Overlapped Windows". A register file based on the Berkeley RISC is incorporated into the ARC.

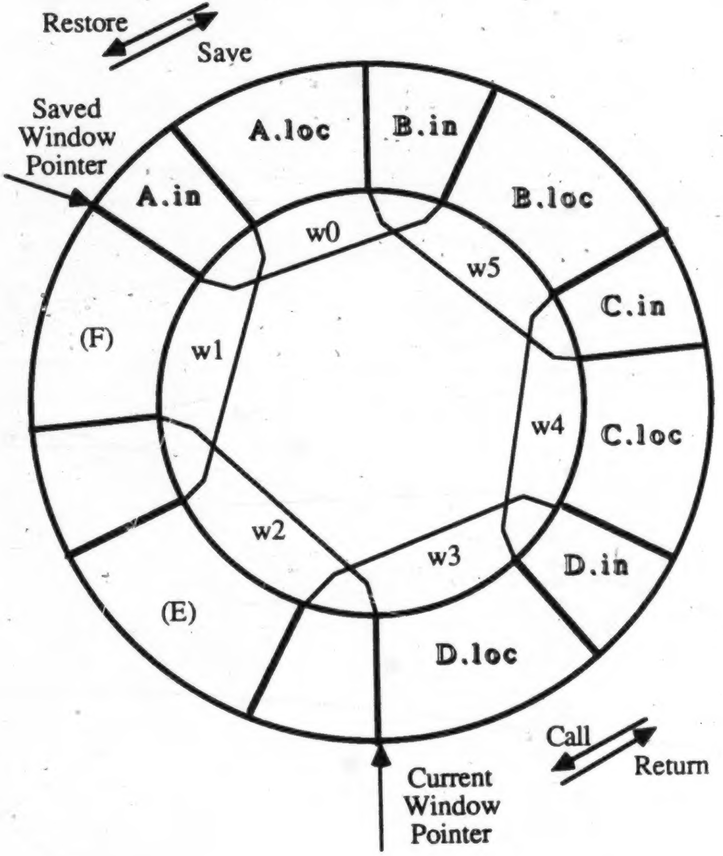
2.4. Pipelining

Most RISC machines utilize pipelining to increase instruction throughput and achieve a high frequency of operation [Stall 88]. Pipelining can be used to speed up the execution of instructions by decomposing instructions into several stages, each of which utilizes a different set of the processor's resources. This results in a greater operating frequency because the critical path of every stage is less than the critical path of the entire instruction. Throughput is maintained if instructions are sufficiently independent with respect to pipeline conflicts and operand dependencies.

In RISC machines, a common instruction processing pipeline consists of the following stages:

a) Calculate next instruction address: Put the program counter onto the address bus.

Resources required: PC, address bus


Restore
Save
Saved Window Pointer
A.loc
B.in
B.loc
A.in
w0
w5
C.in
(F)
w1
w4
C.loc
w2
w3
D.in
(E)
D.loc
Call
Return
Current Window Pointer

Legend:

w0-w5 denote the 6 overlapping windows, each divided into 3 segments. For example, w0 has 10 local registers (A.loc), 6 registers used to transfer input parameters from the calling function (A.in), and 6 registers which are used to transfer parameters to the called function (B.in).

Function Calls
The above diagram portrays register window usage if function A called function B, function B called function C, and function C called function D. Function D is currently executing.

Fig. 2.1: Circular Buffer Organization of Overlapped Windows

b) Fetch instruction: Read the next expected instruction into a buffer.
Resources required: input data bus, instruction buffer 1.

c) Decode: determine the opcode and register operands.
Resources required: decoder, instruction buffer 2.

d) Execute: Perform instruction operation with register operands.
Resources required: ALU, operand busses from registers.

e) Store Result: Store result in register file.
Resources required: bus to register file.

Depending upon the specific processor, the above example pipe stages may be combined to make the pipeline shorter, or further decomposed to lengthen the pipe and increase the clock frequency. Ideally, all pipe stages should have equal delay to maximize efficiency.

The above pipeline will operate smoothly as long as there are no conditional branches and all operands are in registers. Conditional branches are handled in the MIPS and RISC I machines [Henn82] [Kate85] by a "delayed branch" technique. In this technique, the instruction following a conditional branch is always executed to allow time for the conditional branch and next address calculation to clear the pipe.

Practical limits are placed on the number of pipeline stages by the *delayed branch*, and operand dependencies. It is the responsibility of the compiler to reorder the instructions to insure that:

a) operands are prefetched into registers (LoaD Instruction),

b) after each conditional branch there exists an instruction (or NOP), which does not depend on the branch, to keep the pipeline full.

c) operand (register) dependencies are taken into account.

If an operand of an instruction is the result of a previous instruction, idle cycles (NOPs) may be inserted to delay the start of that instruction until the previous one has completed. In general, the higher the number of pipe stages, the more difficult it is for the compiler to create NOP-free code.

2.5. Caching

An I/O bottleneck problem exists in current RISC processors because, according to the standard elements, they ought to execute one instruction per cycle. This means an instruction fetch every cycle which would use all address and data bus bandwidth. Most modern RISC architectures combat this problem by embracing hardware cache support and/or on-chip caches to attain better performance [Furl89]. The caching techniques vary between processors. Intel favors a unified instruction and data cache while Motorola favors separated instruction and data caches (as per the Harvard Architecture) in current and future RISC and CISC processors. Most other RISC chips incorporate an on-chip instruction cache and off-chip data cache to combat the I/O bottleneck problem. The Commercial version of MIPS (MIPS-X) [Horo87] uses both a 2-kbyte on-chip instruction cache and an external interface for high-speed cache access to provide the required memory bandwidth for the processor.

Caching has become necessary to achieve high performance in microprocessor systems. This is particularly so with RISC processors because slower memory cannot supply data at the high clock frequency without wait states given the high I/O bandwidth requirements resulting from an instruction throughput of 1 instruction per cycle. Typical RISC clock speeds have not only increased past the capability of DRAM main memory, but even faster, much more expensive SRAM cannot keep up with RISC processors running above 25 MHz and beyond [Furl89]. Even techniques such as page-mode interleaving for the main memory - where two (or more) banks of RAM may be accessed alternately to reduce the apparent access time - is not good enough because it will cause wait states more frequently than will a cache.

2.5.1. Caching Techniques

Issues in selecting a caching technique include cache placement in the system as well as whether instruction and data caches should be separate or unified.

Placing the cache on-chip or before the address translation of the MMU creates what is called a virtual cache [Furl89]. Because most processors use virtual address locations, a cache that can access data directly from the virtual address without looking up the physical equivalent, will be more effective.

The issue of whether to use a Harvard architecture (divided instruction and data caches) or to use a unified cache is a tradeoff between bandwidth requirements and resource limitations. The Harvard architecture provides an increase in memory bandwidth. The increased memory bandwidth results from an address and data bus to the instruction cache and another address and data bus to the data cache. Therefore, a Harvard architecture machine may have double the bandwidth and also double the number of bus lines and bus pins. A unified cache can encounter more bus contention between data and instruction than a divided cache but a lower pin-count will result.

The ARC may be interfaced to unified or separate instruction and data caches which use virtual addresses.

2.5.2. Overview of Instruction and Data Caches

Instruction caches are normally implemented on-chip because with 1 instruction fetched from cache instead of memory per cycle, the bandwidth saving is high and because the behavior of instructions is well understood as described by [Kate83].:

> Instruction fetches are read-only accesses. They are sequential in small blocks (between *if* or *call* or *loop* statements). Locality arises from the repeated accesses to instructions inside loops. Since programs spend most of their time in small inner loops, this locality is hig

The above behavior allows for a simple instruction cache design.

Data caches are generally larger and more complex than instruction caches and therefore usually implemented as part of an off-chip MMU due to silicon area restrictions. Data memory references are not as predictable as instruction referencing. Locality arises from repeated accesses to the same scalar variables or to sequential access to non-scalar variables such as arrays and structures. For a detailed discussion of microprocessor cache architectures, the reader is referred to [Furl89] [Kate83] [Kado87] [Horo87] and [Hunt87].

2.6. Architectural Features of Stanford's MIPS [Henn82,83]

The Stanford MIPS project [Henn82,83] concentrated on obtaining maximum performance by making simultaneous tradeoffs across three areas: hardware, software support, and systems support. After considering possible design tradeoffs, the MIPS designers opted for less hardware and more complex software. Key features of MIPS in which this is apparent include: load/store architectures, packing constants into instructions, omission of condition codes, word-addressed machines, and imposing pipeline interlocks in software (optimizing compiler).

Load/store architectures require that operands be loaded into registers, from which they are processed. Load/store architectures can yield performance increases if frequently-used operands are kept in registers. The MIPS designers therefore used a large number of registers and an efficient register allocation algorithm as part of the compiler.

To further reduce the number of loads from memory to registers, the MIPS designers also incorporated 4-bit and 8-bit constant fields into some instructions. By studying a collection of programs, they determined that a 4-bit constant would cover 70 % of the cases in which a constant was necessary. An 8-bit constant(allowing a character constant) would cover 95 % of the cases.

The MIPS designers argued against the use of condition codes because they are difficult to implement, difficult for compiler writers, and inefficient for conditional control flow breaks. Condition codes are difficult to implement because they are irregular structures, some instructions set the condition code and others do not. This causes additional logic for condition code control and problems with branches in a heavily pipelined machine (pipe may have to clear or be flushed before a branch). Condition codes are difficult for compiler writers, especially in nonorthogonal architectures, because condition codes are side effects of instruction execution. In terms of efficiency, the MIPS designers compared the use of condition codes in conditional control flow with their scheme of conditionally setting the contents of a general purpose register to 0 or 1, and found the MIPS scheme faster. One notable exception to this scheme was the overflow flag which generates a processor interrupt in the MIPS.

Word-based addressing is advocated by the MIPS designers because it has a lower overhead associated with each fetch or store, and word references occur much more frequently than byte references. To make a word-based approach feasible, special support for accessing bytes was provided in the MIPS instruction set.

Finally, and perhaps most significantly, the MIPS designers advocate shifting the burden of cost from the hardware to the compiler. The MIPS design depends upon a more complex compiler capable of imposing pipeline interlocks and delayed branches by reordering instructions and insertion of NOPs. This resulted in much simpler and much faster hardware. This shifting of the complexity from hardware to software has several major advantages as described by [Henn82].:

> The complexity is paid for only once during compilation. When a user runs his program on a complex architecture, he pays the cost of the architectural overhead each time he runs his program. It allows the concentration of energies on the software, rather than constructing a complex hardware engine, which is hard to design, debug, and effectively utilize.

2.7. Architectural Features of the Berkeley RISC II

The designers of the Berkeley RISC [Patt82] [Kate 83] analyzed the behavior of high level language (HLL) programs, patterns of procedure calls and returns, as well as the optimization of register usage. In addition, they studied the trade-offs between size, complexity, and speed to obtain the most effective use of the scarce hardware (silicon) resource in the execution of HLL programs.

The Berkeley RISC project began in the spring of 1980. The RISC I processor was implemented in the fall of 1980. Additions to the instruction set and changes to the on-chip organization/communication resulted in the RISC II by spring 1983.

The RISC II has 39 instructions. These instructions are subdivided into 4 categories:

- 12 ALU instructions
- 16 Memory Access (Load/Store) instructions
- 7 Branch and Call instructions
- 4 Miscellaneous Instructions

All instructions are 32 bits in size. Most instructions contain 3 operands with 2 sources and a destination register specified in the instruction. There are also some two- and single-operand (address) instructions. For memory access instructions, there are two addressing modes: a) indexed (contents of register + immediate offset), and b) PC relative (PC + immediate offset).

The RISC II has 138 32-bit working registers available to the user. Each procedure can access 32 of these registers. The first 10 registers, R0, R1, ..., R9, are *global registers* and are always accessible. The other registers are called *window registers*. 22 of these registers, R10, R11, ..., R31, are accessible within a procedure. Ten of the registers, R16 to R25, within each *register window* are *local registers*. Registers R10 to R15 are used to hold parameters passed by the current procedure to a called procedure. Registers R26 to R31 hold parameters passed to the current procedure from the procedure which called it.

There are 8 windows in the RISC II register file. The current window is indicated by the 3-bit Current Window Pointer (CWP) of the Program Status Word (PSW). When a procedure call occurs, the CWP is decremented (mod 8). When a procedure return occurs, the CWP is incremented (mod 8). To support the register organization described above, the register windows are organized in a circularly overlapping fashion so that all register windows have 6 registers in common with each of the adjacent register windows. The registers which overlap are used for parameter passing between procedures. The circularly overlapping structure of the RISC multi-window register file is shown in Fig. 2.1.

The most recently used window of the register file which has been saved to memory is indicated by the 3-bit Saved Window Pointer (SWP) of the PSW. When a procedure call occurs so that CWP would become equal to SWP, a *register file overflow* occurs. Similarly, when a procedure return occurs so that the CWP would become equal to the SWP, a *register file underflow* occurs. These underflow and overflow traps essentially occur when there is no more space in the register file. When a *register overflow* occurs, a register window is saved to memory and when a *register underflow* occurs, a register window is restored from memory. Tamir and Sequin [TaSe83] have investigated this aspect of the RISC architecture and concluded that the best strategy is to save only one window per overflow trap. In the ARC processor, improvements on the above scheme for saving and restoring registers have been made. In the ARC, register windows are saved and restored in the *background* without halting or a context switch to an interrupt service routine. The *background* register save/restore concept will be explained further in chapter 3.

The RISC II instruction pipeline has three stages: a) Fetch, b) Compute, and c)

Write. As well, a delayed branch feature has been incorporated into the RISC II. This means that the jump or branch takes effect only after the instruction following the branch has been executed. Another restriction of the RISC II pipeline was that the pipeline was suspended during data memory accesses. This suspension occurred because the address and data busses are used for the data load or store thereby disallowing instruction fetches to occur (which also require the address and data busses). As well, the RISC II register file allows only one register-write per cycle. This means that a dummy pipeline stage must be inserted into *all* instructions at the place where loads perform their memory access. An on-chip instruction cache would combat the problem of pipeline suspension during data access as long as a cache miss does not occur. Pipelined memory access (more than 1 memory access in progress at a time) combined with dual-ported memory would also alleviate the problem.

2.8. Summary of Previous Research

The ARC architecture incorporates many standard RISC characteristics:

- A limited and simple instruction set.
- A large number of registers.
- Load and store access to memory.
- All instruction operands in registers.
- A strategy of maximizing the use of registers and minimizing references to memory.
- The use of an optimizing compiler
- An emphasis on optimizing the pipelined execution of instructions to obtain a throughput of 1 instruction per cycle.

In addition to the standard RISC elements, characteristics from Stanford's MIPS [Henn82,83] and the Berkeley RISC [Kate83] [Patt82] were incorporated into the ARC architecture.

In particular, these characteristics include the following:

a) circular, overlapping register windows. [Kate83] [Patt82]

b) non-interlocked pipeline, optimizing compiler [Henn82,83]

c) delayed branches [Henn82] [Kate83]

d) lack of condition code [Henn82]

Most RISC processors require caching techniques to obtain sufficient I/O bandwidth and an instruction throughput of 1 instruction per cycle. This is evident in the commercial versions of the RISC II [Kate85], and the MIPS-X [Horo87], both of which require an on-chip instruction cache and off-chip data cache. The more aggressive Motorola 88000 RISC chip utilizes a Harvard architecture cache structure to facilitate high I/O bandwidth. ARC employs I/O bandwidth reduction to eliminate the need for an on-chip cache (as long as the memory is fast enough). If memory is not fast enough, ARC will require a cache structure. Separate instruction and data caches could be used to afford some advantages of the Harvard architecture with fewer bus lines and pins. If the process technology permits, the cache(s) together with other memory management functions can be moved on-chip. The novel concepts of the ARC which increase performance are presented in the next chapter.

3. NOVEL CONCEPTS THAT INCREASE SPEED AND EFFICIENCY

A number of novel concepts have been introduced with ARC which afford improvements over previous RISC designs. These concepts include:

a) 2 instructions per 32-bit word

b) unidirectional input and output busses

c) background register saves

d) pipelined memory access

e) external program counter and associated logic

f) memory control unit (MCU) or *intelligent memory*

The improvements are realized in terms of greater I/O bandwidth available, fewer nonprocessing states, and the ability to access slow memory. Greater I/O bandwidth is available because concepts a), c), e) and f) cause a reduction in the total bandwidth required for instruction fetches and subroutine calls, while concepts b), c), e), and f) are used to balance the bandwidth between the I/O resources available. The ARC has fewer nonprocessing states because of the MCU and *background* register saves. The ARC may also be interfaced with slow memory by pipelining memory access.

The increase in available input bandwidth results largely from packing two instructions per word, and from the unidirectional IN bus. The increase in available output bandwidth results largely from the external PC (instruction addresses do not originate from ARC) and from the unidirectional OUT bus. Conventional and RISC processors normally increase their available bandwidth by having on-chip instruction and/or data caches. This works well for traditional programming languages which display good locality properties. In comparison, techniques to be detailed in this chapter for increasing ARC's available bandwidth do not rely upon program locality and thus should provide better support for languages such as LISP and Smalltalk. Nonetheless, the ARC techniques can also operate in conjunction with a cache structure to gain the normal benefits of caches.

Background register saves and restores are made possible by the large amount of available bandwidth. This feature allows procedure calls and returns to occur with a minimum of LoaD and STore overhead associated with restoring/saving register windows.

The division/interface between ARC and the MCU (or intelligent memory) is also a key concept which results in a desirable division/interface between processor and memory. This interface between ARC and the memory system forms the basis of a set of philosophical objectives detailed at the end of this chapter.

3.1. Two Instructions per 32-bit Word

The available input bandwidth is significantly increased by encoding most ARC instructions into 16 bits. Thus two instructions are loaded in most 32-bit instruction fetches. As the ARC executes 1 instruction per cycle, the unidirectional IN bus is available for data transfers every second cycle. This reduces the I/O bottleneck, leaving bandwidth on the IN bus available for incoming data or register restores. In addition, halfword instructions are easier to decode and should result in less memory and cache space required for executable code than would fullword instructions. The reduction in cache space required is an important feature because cache memory is always at a premium if it is to be implemented on the same IC as the ARC or the MCU.

To gain the above advantages, several tradeoffs were made. Because the instructions are restricted to 16 bits in length, the instruction set is limited and slightly less orthogonal than it might be otherwise. As well, the register operand fields are only 4 bits wide so that a maximum of 16 registers can be addressed. Therefore the register windows contain 16 registers and only 1 16-register window is available to each procedure. These disadvantages are not serious because the instruction set is adequate and follows the RISC philosophy and the register window size is only 6 smaller than that of the Berkeley RISC II but missing 10 global registers. The advantages of greatly reducing the I/O bottleneck and of reducing memory and cache requirements should outweigh these disadvantages.

3.2. Unidirectional I/O

Conventional microprocessors have a unidirectional address bus and a bidirectional data bus. In conventional microprocessors, the address bus is used for both instruction addresses and data addresses, while the data bus is used for instructions, incoming data and outgoing data.

Higher performance microprocessors with high I/O requirements such as the Motorola 68030 and 88000 follow the Harvard architecture scheme. This is a four bus architecture having an instruction address bus, a data address bus, an instruction input bus and a bidirectional data bus. The Harvard architecture eliminates conflicts between instructions and data at the expense of additional pins and busses.

In the ARC, outgoing data addresses and data utilize the OUT bus while instruction pairs and incoming data utilize the IN bus. This I/O structure provides equal input and output bandwidth resources to support the almost equal input and output bandwidth demand which results when instruction addresses do not originate from the cpu. The ARC processor has certain similarities with Harvard architecture processors, using time division multiplexing (TDM) to eliminate the high pin count. The ARC architecture may be considered as a TDM Harvard architecture with respect to the IN bus because the IN bus is reserved for instructions and data on even and odd clock phases respectively. The ARC can also be interfaced to separate instruction and data caches as is done in the Harvard architecture. ARC does not require as much I/O bandwidth as is available with the Harvard architecture because of the features which reduce the necessary I/O bandwidth requirements. Conflicts between instructions and data are resolved by pipelining as explained in sections 3.4 and 3.5.

ARC has higher input bandwidth available than a conventional RISC. This is because ARC's IN bus supports incoming instruction pairs and incoming data. In comparison, the conventional data bus must support outgoing data in addition to incoming full-word instructions and incoming data. ARC also has a higher output bandwidth available for data transfer. This is because ARC does not need to send out an instruction address on each cycle as is necessary with conventional processors.

Further, the traditional bus structure of most RISCs incorporates a bidirectional bus. The associated bus *turn-around* delay and clock skew may reduce the maximum operating frequency. Utilizing unidirectional busses in ARC eliminates bus turnaround delays and makes complicated bidirectional drivers/receivers between the processor and memory unnecessary. [1]

As well, unidirectional busses may make it easier to design a pipelined memory subsystem, impossible in a conventional system with a bidirectional data bus. The ARC's unidirectional bus system eliminates bus conflicts between data to be stored and data to be loaded.

3.3. Background Register Saves and Restores.

To reduce the number of nonprocessing states, the ARC design supports *background* register saves and restores. The ARC includes a register file made up of circular overlapping register windows, similar to those developed in the Berkeley RISC [Patt82]. During procedure calls, a new register window becomes the "active" window. The "active" window holds the only registers that can be accessed by ARC instructions. To reduce the number of wait states (LoaD or STore instructions) associated with restoring or saving the window context to a register stack, the ARC incorporates additional logic which saves and restores register context windows in parallel with ALU operations. Whenever the IN bus is not being used, this logic checks register context windows and will restore the context of ancestors of the currently active window in anticipation of subroutine *returns*. Similarly, when the OUT bus is not being used, this logic will attempt to store the context of a higher register window to the register context stack in anticipation of a subroutine *call*. This feature is used to make it seem like there is an infinite number of windows in the register file. If a number of *calls* or *returns* occur quickly in succession, it is possible that the background saving and restoring will fall behind. The MCU will detect this condition and the processor will be fed NOPs so that the registers will be saved/restored at the rate of one per cycle. [2]

---

[1] Tristate drivers would still be required to support DMA.

ALU instructions do not utilize the IN, OUT, or MMU busses. Therefore, while the processor is executing ALU instructions, the OUT and MMU busses are free and the IN bus is free every odd phase (instruction fetch on even phases). This available I/O bandwidth is exploited to save and or restore registers without interfering with the normal instruction execution throughput. Because this process occurs while ALU instructions execute, it is referred to as a *background* save or restore of registers.

3.4. Pipelined Memory Access and Instruction Execution

Pipelined memory access allows more than one memory access to occur at the same time in a pipelined fashion. Pipelined memory access reduces the number of overhead states and allows a higher operating frequency. In a traditional RISC approach, the MMU must decide if data is resident (in cache or memory) and return the data within a specified time to the cpu (typically 1 cycle). When the data is not resident, the processor is halted until the data is fetched. Further, the time required to determine if the data is resident affects the maximum operating frequency of the system. The ARC incorporates a pipelined memory access scheme. For LoaD instructions, addresses are sent out in one cycle and the data is expected to be returned N cycles later. This allows the external MCU to take several cycles to locate the requested data and return it to the cpu.

---

[2] This achieves the same effect as would a burst mode Load/Store of the register window.

Further, N can be dynamically adjusted [3] to support complex memory decoding. For example, when running LISP or object-oriented programs, the pipeline length can be increased to allow time for the MMU to accomplish translation of symbolic program addresses to physical addresses without inserting wait states. The Pipeline is extended to the Memory Control Unit because transport and control instructions are pipelined. In the first version of ARC, Load and Store instructions send out the memory address and expect the data to be transferred to or from ARC, on the next phase. In the case of JMP or JSR, the new instructions are not fed to ARC until 2 phases after the memory address is sent out.

Caching and memory interleaving can be used to allow ARC to work at a significantly higher speed than the main memory. Caching is favored by most high performance processors because very high hit ratios can be achieved with a moderately sized cache and the hit ratio can be increased by increasing the size of the cache. Page mode memory interleaving (Least significant address bits select bank) can also be used when sequential memory locations are accessed. This allows different banks of memory to be accessed alternately to increase the memory transfer throughput in a pipelined fashion. Page mode interleaving has not proven to be as successful as caching because often the same memory bank is accessed twice in succession.

The pipelined memory access described above is beneficial because it allows the processor to interface with slower RAM. The number of pipe stages could conceivably be changed depending upon the memory subsystem (slower memory or to take into account cache misses).

---

[3] The maximum size of N would, however have to be known at compile time and it is likely that less efficient code will result as the ceiling of N increases.

3.5. External Program Counter

Locating the program counter (PC) off-chip significantly increases the output bandwidth available for data transfers. In the ARC, the instruction address does not originate from the cpu, but rather, from a memory control unit (MCU) (except in the case of a branch). Because most instruction addresses are externally generated and stored, the instruction address requires little of ARC's OUT bus bandwidth. By monitoring the instruction stream for branch instructions and receiving the result of branch conditions from the ARC cpu, the MCU could also generate branch destinations.

Other advantages of an off-chip PC include an increase in parallelism, a simpler on-chip pipeline, and locality of instruction address generation. Associated with the external PC is some logic to allow instruction address sequencing, addition, etc. By having the PC and associated logic off-chip, on-chip resources such as the ALU are not used to update the PC. The instruction pipeline of ARC is simplified because next instruction calculation does not have to fit into the pipeline resource scheduling. By associating the PC more closely with memory, there exists a greater locality of instruction address generation. Thus, instruction addresses are not propagated to the processor and calculated there. Rather processor addresses are calculated locally within the *intelligent* memory subsystem (or MCU) and only the instructions are fed to the processor.

It is possible to locate the PC off-chip as part of the MCU because ARC has simple addressing modes and control instructions. (See chapter 4 for details of the ARC instruction set.)

During sequential instruction flow, the external PC simply increments. Otherwise, the external PC unit needs to be notified when JUMP, Jump to SubRoutine, RETurn, and relative BRanch control instructions occur. When a JUMP instruction occurs, the contents of a register are placed on the OUT bus to become the new PC. When a JSR instruction occurs, the PC is saved to the PC stack, and the contents of a register are placed on the OUT bus to become the new PC. When a RETurn instruction occurs, the last value on the PC stack becomes the new PC. When a relative BRanch instruction occurs, the 8 least significant bits on the OUT bus are added (2's complement) to the current PC.

ARC has 2 addressing modes: a) register direct and b) immediate. Therefore all data memory addresses originate from the general purpose registers. PC relative and other addressing modes are synthesized by loading the PC (or other special purpose register) into a general purpose register, performing an arithmetic operation on the register (such as ADD constant), and then using the resulting address in a LoaD or STore.

3.6. Intelligent Memory

The Memory Control Unit (MCU) may be considered as a single chip unit or as several chips distributed within the *intelligent* memory subsystem. The MCU works in conjunction with ARC to enable bandwidth reduction, bandwidth balancing, *background* register saves, and pipelined memory access.

Bandwidth reduction from ARC is made possible, in part, because the PC and other special registers are part of the MCU. By associating the PC more closely with memory, instruction addresses are calculated within the *intelligent* memory subsystem. This leads to saved bandwidth because the PC is only sent out on branch instructions. An external PC also results in an increase in parallelism and locality of instruction address generation.

Associating most other special registers with the MCU leads to further bandwidth reduction and locality of interrupt handling and context switching. When an interrupt occurs, it is unnecessary for ARC to transfer all special registers to the MCU, thereby saving bandwidth. Interrupts and context switches are primarily the responsibility of the MCU. (ARC treats interrupt service routines as just another subroutine call.) Having the MCU handle interrupts makes sense because most interrupts result from memory and I/O devices and can thus be handled more locally. Having the MCU handle context switches also makes sense because a context switch involves memory reallocation and sometimes disk swapping as a new process becomes active.

Bandwidth balancing is made possible by the interface between ARC and the MCU. When the PC is part of the MCU, nearly equal IN and OUT bus bandwidth requirements exist to and from ARC. The OUT bus of ARC supports outgoing data, data addresses, and register saves. The IN bus of ARC supports instruction pairs, incoming data, and register restores. Bandwidth must also be balanced between phases of Phi1 and Phi2 as hinted in section 3.2 in the paragraph on time-division-multiplexing (TDM) of the IN and OUT busses. Bandwidth balancing is explained more fully in terms of resource scheduling in section 4.1.4 and pictorially in Fig. 4.8.

Background register saves are also made possible by the MCU. The MCU manages the register stack. The algorithm to have registers saved to (or restored from) the register stack is a responsibility of the MCU. Storage space for the registers is also a responsibility of the MCU. When the MCU determines that it is time to either save or restore a register subwindow, it notifies ARC. Then during each subsequent ALU instruction, a register from the subwindow to be saved/restored will be *push*ed onto/*pop*ed from the register stack.

Pipelined memory access is another useful feature of the interface between ARC and the MCU. The MCU may control slow memory, utilize virtual address translation, utilize caching, or queue addresses to interleaved memory. In these cases, the number of cycles between the assertion of a data address from ARC and the response from memory can be customized for each different type of memory subsystem.

3.7. Philosophy Behind ARC and MCU Design

In addition to the concepts presented in this chapter to improve speed and efficiency, a number of other high level objectives are set for the design of ARC and its companion MCU.

The ARC and MCU are partitioned such that generic processing functions are incorporated into ARC while memory control functions are incorporated into the MCU. The implementation of the processing and memory control functions on separate chips have a number of advantages:

a) It allows the processor to be generic and attached to different, specialized memory controllers. Memory controllers tend to be dependent upon the system architecture, memory subsystem hardware, the operating system, etc. In contrast, processors should be as generic as possible so that they can be incorporated into a wide variety of systems and the executable code will be the same on all systems.

b) Fewer compromises are necessary with this partition than if memory and processor functions are incorporated together on the same chip (e.g. number of registers/bits vs. size of cache, more transistors vs. lower yield).

c) A sibling relationship between cpu(s) and memory subsystem(s) results in increased parallelism and locality of information processing. Conventional processors exhibit a master-slave relationship between the cpu and the memory subsystem. The MCU thus operates in parallel with ARC by performing memory control functions and feeding instructions and routing data to and from ARC. Functions such as caching, interrupt handling, context switches, and virtual memory translation are functions of the MCU because these functions are associated more closely with memory control than data processing. By giving the MCU these responsibilities, some of the processing and bandwidth load are taken from the processor.

d) More efficient use of I/O bandwidth as well as locality of information processing and an increase in parallelism results from incorporating special registers such as the PC into the MCU. Because instructions originate from memory and the generation of instruction addresses requires little CPU intervention, the PC and associated sequencing logic is implemented in the MCU. Other special registers such as the Register Stack Pointer are associated with the MCU because they address memory locations used by memory control functions. Because these special registers do not have to be saved and restored to ARC at the advent of an interrupt, interrupt handling becomes the responsibility of the MCU. This leads to locality of interrupt handling because many interrupts originate from the memory subsystem(s) (e.g.: page faults, segmentation faults); It also leads to increased parallelism because the MCU takes responsibility for feeding ARC the call to the correct interrupt service routine.

e) The ARC design is versatile and expandable. On-chip enhancements could include more registers, different register window overlaps, more sophisticated I/O Control Unit (IOCU) functions, different ALU operations, and possibly expansion to 64 bits. The Memory Control Unit would be closely bound to the system and memory architecture. The MCU could be a single chip or distributed control in the memory subsystem(s). Enhancements to the MCU could include larger caches, virtual memory, tags and security features. In effect, both ARC and the MCU can be "scaled" according to the available technology.

The architectural relationships between ARC and the MCU will be discussed in more detail in the next chapter.

4. ARCHITECTURE

This chapter presents a description of the architecture of the ARC processor at the block diagram or functional level. Architectural details of the ARC processor include datapath functions, I/O Control Unit (IOCU) functions, the instruction set, and the instruction pipeline. The implications of the ARC architecture on the system level design are then discussed. ARC system architectural details include the interface of ARC to different types of memory subsystems. These discussions will highlight the implementation details of the novel architectural concepts and philosophical issues presented in Chapter 3.

4.1. ARC Processor Architecture

A picture of ARC is shown in Fig. 4.1. The 2 major functional blocks of ARC are the Datapath and the IOCU. The Datapath is composed of the ALU and Register File while the IOCU handles background operations and pipelining as well as I/O operations. The relative positions of the functional blocks and I/O signals are detailed in Fig. 4.2. Note the nonstandard unidirectional IN and OUT busses, interface to the external MCU, and 4 nonoverlapping clock phases. [1]

4.1.1. Datapath

A Block diagram of the Datapath is shown in Fig. 4.3 The Datapath is composed of the ALU and the Register File. The ALU communicates with the register file via 2 read ports and 1 write port. The Register File communicates with the outside world via the IOCU.

---

[1] Clock phases are divided into *even* and *odd*, as well as *A* and *B* clock phases. Even clocks include Phi0A and Phi0B. Odd clocks include Phi1A and Phi1B. During A phases, busses are precharged, the ALU writes to the register file, and latching occurs. During B phases, all decoding, execution, memory access, etc. occurs.

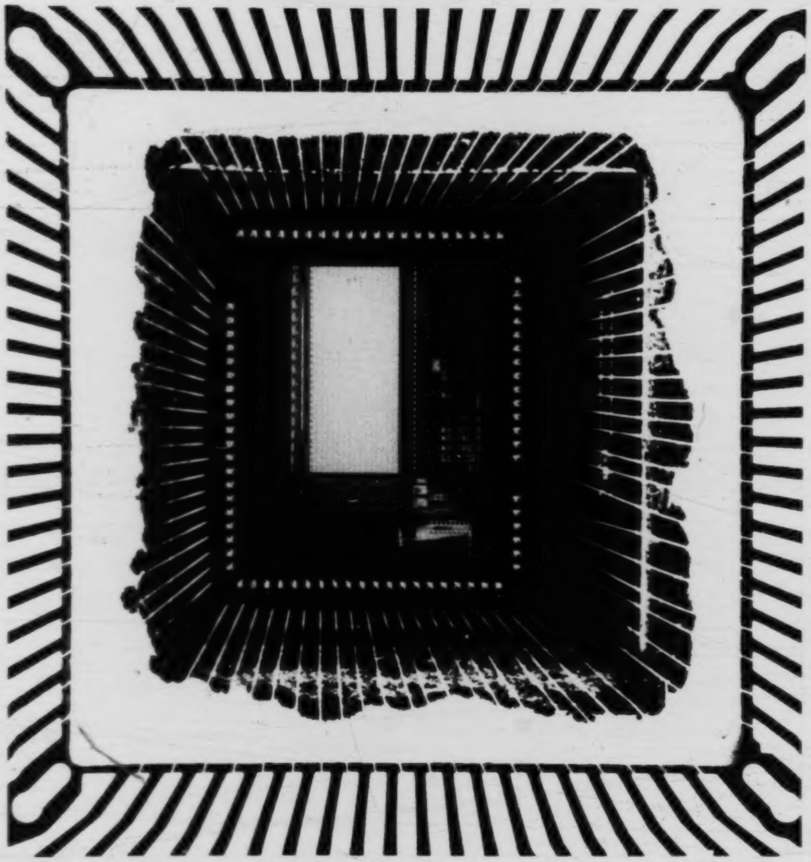
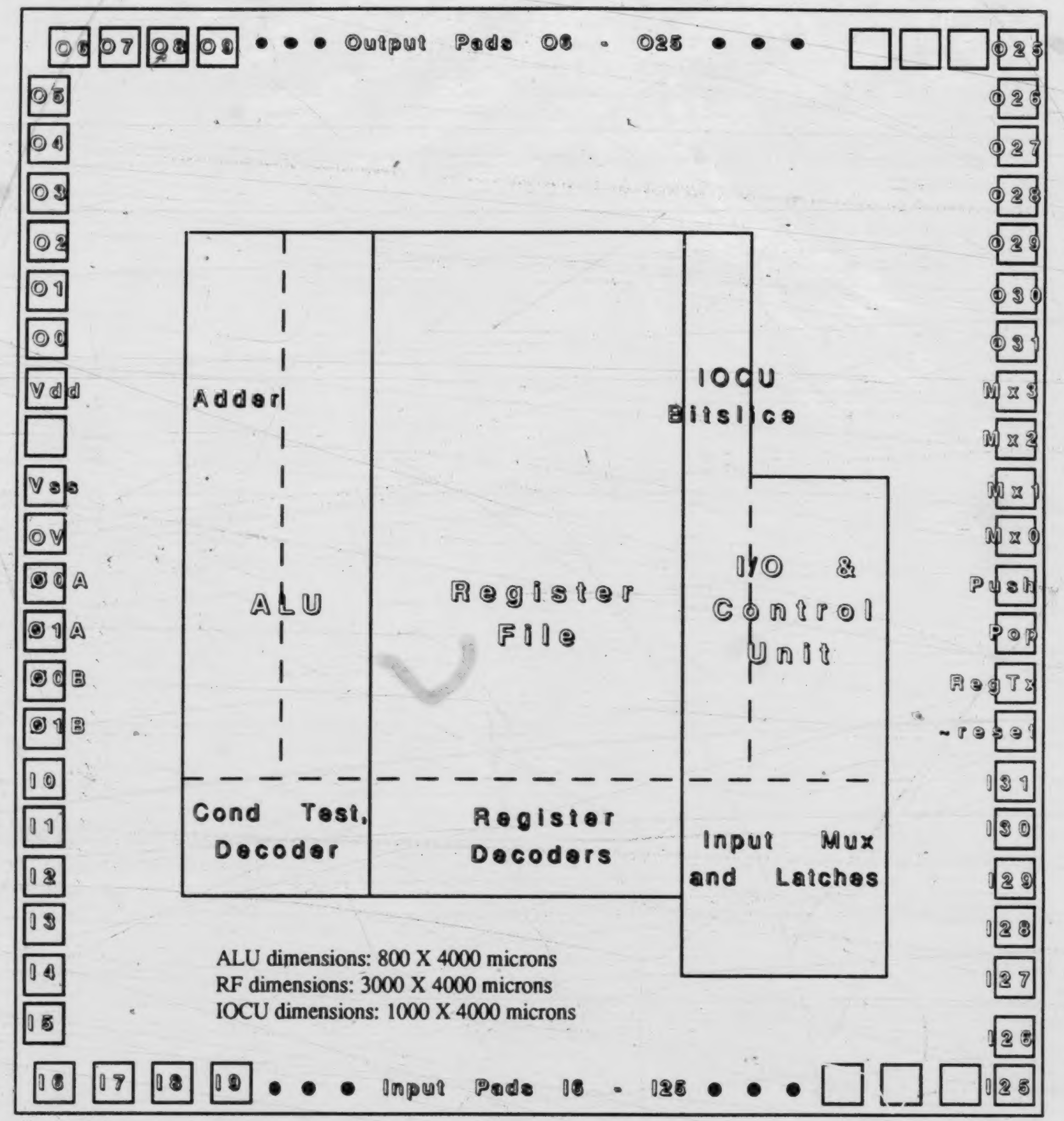


Fig. 4.1: Photograph of ARC Die


O6
O7
O8
O9
• • • Output Pads O6 - O25 • • •
O25
O5
O4
O3
O2
O1
O0
Vdd
Vss
OV
Ø0A
Ø1A
Ø0B
Ø1B
I0
I1
I2
I3
I4
I5
O26
O27
O28
O29
O30
O31
Mx3
Mx2
Mx1
Mx0
Push
Pop
RegTx
~reset
I31
I30
I29
I28
I27
I26
Adder
ALU
Register File
IOCU Bitslice
I/O & Control Unit
Cond Test, Decoder
Register Decoders
Input Mux and Latches
I6
I7
I8
I9
• • • Input Pads I6 - I25 • • •
I25

ALU dimensions: 800 X 4000 microns
RF dimensions: 3000 X 4000 microns
IOCU dimensions: 1000 X 4000 microns

Legend:

I0-I31: IN Bus/Input Pads
O0-O31: OUT Bus/Output Pads
OV: Overflow Flag (from ARC to MCU)
Mx0-Mx31: MCU Opcode (" ARC TO MCU)
Push,Pop,RegTx,~reset: Background Register
    Save/Restore Control Signals (MCU to ARC)

Ø0A: Clock used for even-phase precharge and latching
Ø0B: Clock input used for even-phase execution
Ø1A: Clock used for odd-phase precharge and latching
Ø1B: Clock used to synchronize odd-phase execution

Fig. 4.2: ARC Die Floorplan

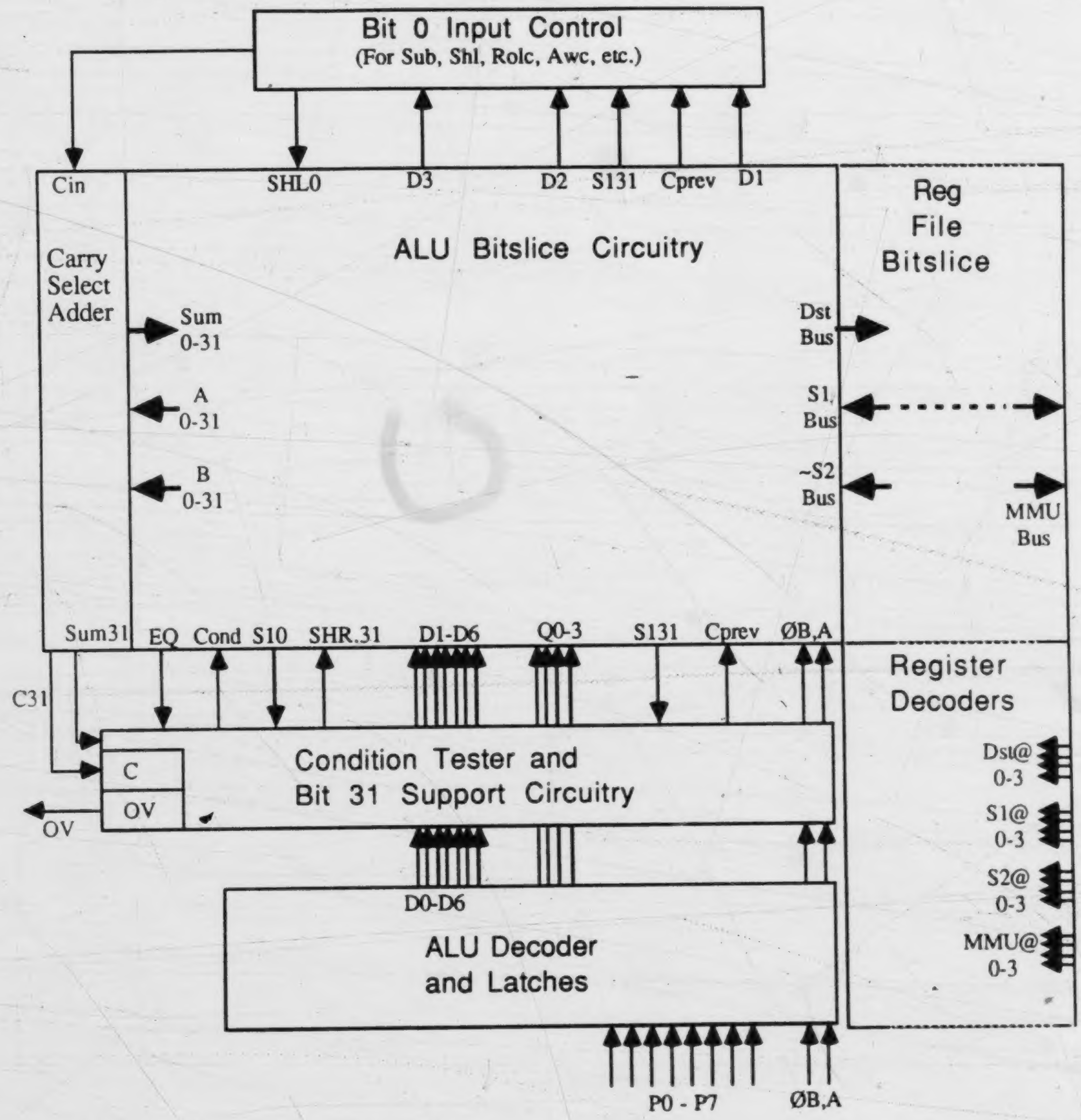

Bit 0 Input Control
(For Sub, Shl, Rolc, Awc, etc.)
Cin
SHL0
D3
D2
S131
Cprev
D1
Reg File Bitslice
ALU Bitslice Circuitry
Carry Select Adder
Sum 0-31
A 0-31
B 0-31
Dst Bus
S1 Bus
~S2 Bus
MMU Bus
Sum31
EQ
Cond
S10
SHR.31
D1-D6
Q0-3
S131
Cprev
ØB,A
Register Decoders
C31
Condition Tester and Bit 31 Support Circuitry
C
OV
OV
Dst@ 0-3
S1@ 0-3
S2@ 0-3
MMU@ 0-3
D0-D6
ALU Decoder and Latches
P0 - P7
ØB,A

Fig. 4.3: ALU Block Diagram

4.1.1.1. The ALU

The ALU is shown as the left half of the Datapath Block Diagram of Fig. 4.3. The ALU is responsible for operations on register and instruction operands. The result of each ALU operation is stored in the register file. A carry-select adder, condition testing logic, and an ALU instruction decoder are incorporated into the ALU. The interface and relative placement of these blocks of the ALU are illustrated in Fig. 4.3. The architectural design of the ALU is fairly conventional. The instructions supported by the ALU are presented in Section 4.1.3.1 and the logic design of the ALU is presented in Chapter 5.

4.1.1.2. The Register File

The register file is responsible for storing the working set of data to be operated upon by the ARC processor. In any LoaD-and-STore architecture, the design of the register file greatly influences the throughput of the processor. Previous RISC processors have incorporated register files as large as possible given the physical constraints of the implementation technology. With a large register file, more of the working set of data can be held on-chip, causing fewer LoaD and STore accesses to memory. In a conventional RISC, memory accesses are costly in terms of bandwidth, data latency, and pipeline conflicts (e.g.: interference of LoaD and STore with Instruction Fetch in Berkeley's RISC).

The register file is shown as the right half of the Datapath Block Diagram of Fig. 4.3. The main difference with regard to previous register file designs is that four busses are served by the ARC Register File: S1, ¯S2, Dst, and MMU. The S1 bus carries the contents of a register to the ALU and the IOCU. The ¯S2 bus carries the contents of a register to the ALU. The Dst Bus carries the result of ALU operations back to a register. The MMU bus is bidirectional and is used for data transfer between a register and the IN and OUT busses via the IOCU. The register addresses for each of the 4 busses are input to the subwindow decoders and are labeled Dst@, S1@, S2@, and MMU@ in Fig. 4.3.

This high connectivity to the register file allows more independence of operation between the ALU and the IOCU thereby simplifying *background* register saves/restores, simple pipeline scheduling, and high instruction throughput. *Background* register saves/restores or memory access instructions can occur in ARC while an ALU operation is in the pipe because sufficient busses exist to access the register file concurrently.[2]

The Register File is organized as a set of overlapped register windows. The advantages of a circular buffer organization of Overlapped windows was shown by the Berkeley RISC group [Patt82] and was mentioned in Chapter 2. Essentially, register windows allow the exploitation of temporal and spatial locality of address and data references within a procedure and across procedure calls. Because the register windows are overlapping, some registers accessed by the calling subroutine are also accessible by the called subroutine. This enables efficient parameter passing.

The version of the ARC implemented had 32 registers in total, divided into 4 subwindows of 8 registers each. 2 subwindows are accessible at any one time, 1 of the subwindows would previously have been accessible to the calling procedure resulting in an overlap of 8 registers between procedure calls. (A more detailed physical description of the ARC register file circuitry is given in Section 5.2)

This circular buffer organization of overlapped register windows leads to a register stack structure as depicted in Fig. 4.4. The register stack is essentially the register window of old procedure activations stored in memory because of limited space within the register file.

---

[2] In a processor architecture with one less bus to the register file, background register saves could still occur by using the idle bus during 2 operand ALU instructions and branches. However, this would reduce the opportunity to perform background saves and restores resulting in more waits required to keep pace with subroutine calls and routines.

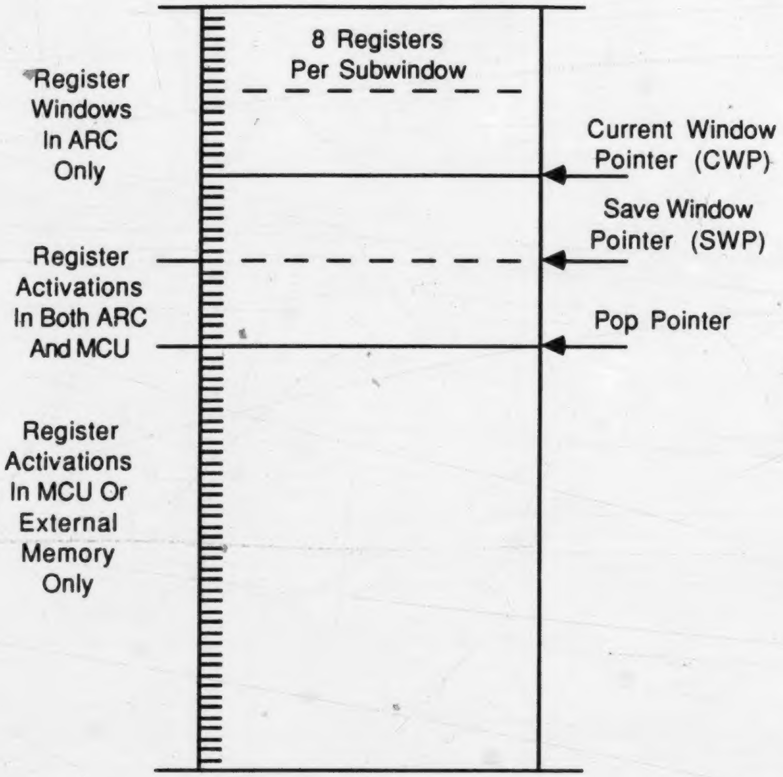

8 Registers
Per Subwindow
Register
Windows
In ARC
Only
Current Window
Pointer (CWP)
Save Window
Pointer (SWP)
Register
Activations
In Both ARC
And MCU
Pop Pointer
Register
Activations
In MCU Or
External
Memory
Only

Fig. 4.4: Register Stack

The register stack is controlled by the MCU and the *background* register save/restore finite state machine. The MCU keeps track of procedure calls and returns and interrupts and notifies ARC if a register subwindow *push* or *pop* is to occur. The current register window is referenced by the Current Window Pointer (CWP). A *background* register save *pushes* the register subwindow referenced by the Save Window Pointer (SWP) A *background* register restore *pops* the register subwindow preceding the pop pointer. As shown in Fig. 4.4, only the current and most recent procedure register sets are held in the ARC register file. The register sets of distant ancestors of the current procedure are stored only in memory controlled by the MCU except when the register stack is nested less than two levels deep in which case all procedure activations are in registers (e.g.: at boot)

4.1.2. The I/O and Control Unit (IOCU)

A block diagram of the IOCU is shown in Fig. 4.5. The IOCU interfaces the register file to the IN and OUT busses and the MCU. The IOCU may be considered as being composed of four functional blocks:

1) I/O Bitslice Circuitry (upper left of Fig. 4.5)
2) Instruction Extraction and Latching Circuitry (bottom of Fig. 4.5)
3) Transport and Control Instruction Circuitry (upper right of Fig. 4.5)
4) Window Control Logic (mid right of Fig. 4.5)

An overview of each of the four functional blocks follows. For more detailed information about the IOCU circuitry consult Section 5.3.

The I/O Bitslice Circuitry is responsible for routing of data between the IN bus, the OUT bus, and the S1 and MMU busses of the register file. This circuitry facilitates a high bandwidth interface between the Register file and the unidirectional IN and OUT busses, providing support for the routing of data, necessary for transport and con-

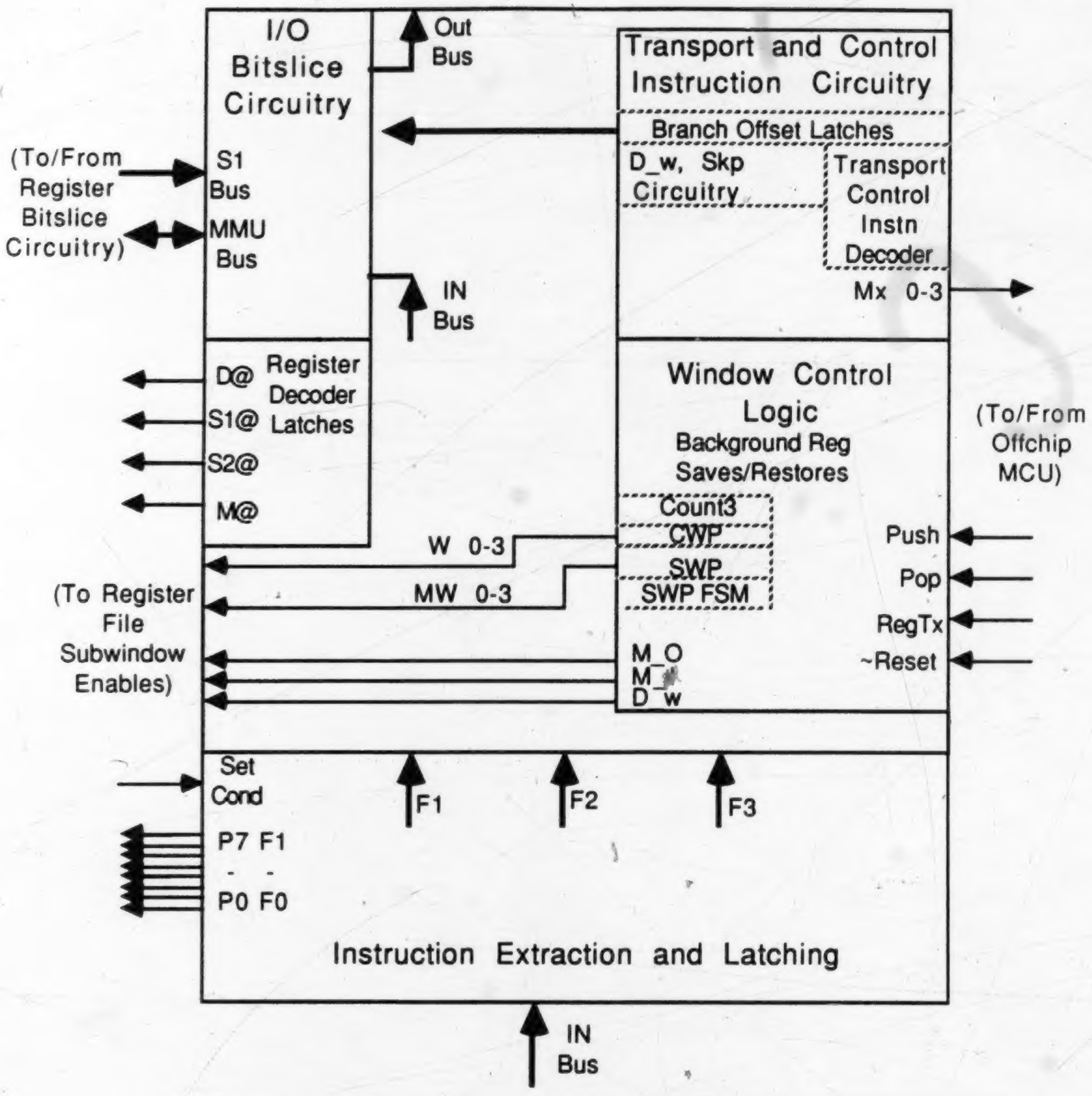

I/O Bitslice Circuitry
Out Bus
Transport and Control Instruction Circuitry
Branch Offset Latches
D_w, Skp Circuitry
Transport Control Instn Decoder
Mx 0-3
(To/From Register Bitslice Circuitry)
S1 Bus
MMU Bus
IN Bus
Register Decoder Latches
D@
S1@
S2@
M@
Window Control Logic
Background Reg Saves/Restores
Count3
CWP
SWP
SWP FSM
(To/From Offchip MCU)
Push
Pop
RegTx
~Reset
W 0-3
MW 0-3
(To Register File Subwindow Enables)
M_O
M_
D_w
Set Cond
P7 F1
P0 F0
F1
F2
F3
Instruction Extraction and Latching
IN Bus

Fig. 4.5: I/O and Control Unit Block Diagram

trol instructions as well as *background* register saves and restores.

The Instruction Extraction and Latching Circuitry is responsible for the extraction of opcodes and instruction fields and latching the instructions into the pipeline. The extraction of instructions is necessary because ARC instructions are packed 2 per 32-bit word (as discussed in Chapter 3). The Instruction Extraction Circuitry divides an incoming instruction pair into a first-halfword instruction and a second-halfword instruction. The first-halfword instruction is "latched" into the pipeline one phase ahead of the second halfword instruction. Once an instruction is latched into the decode phase of the pipeline, the instruction type and operand fields are decoded. The operand fields are passed to the Register Decoder Latches shown above and to the left of the Instruction Extraction Circuitry in Fig. 4.5.

The Transport and Control Instruction Circuitry is responsible for the decoding and pipelining of transport and control instructions as well as interfacing with the external MCU. Decoding of transport and control instructions is performed by a local I/O Control Instruction Decoder. Circuitry to implement the pipelines of the various transport and control instructions is coupled closely with the local decoder. The Transport and Control Instruction Circuitry interfaces to the external MCU via the Mx0-3 signals which inform the MCU of the current transport or control operation. This, in turn, facilitates transfers to and from memory for LoaD and STore instructions as well as background register saves/restores. The interface with external memory will be discussed in detail in sections 4.2, and 5.3.

The Window Control Logic circuitry is responsible for the state of the register stack and the overlapping register windows. The Window Control Logic Circuitry houses the Current Window Pointer (CWP), the Save Window Pointer (SWP) and circuitry to perform background register saves/restores. The background register save/restore circuitry is governed by the Push, Pop, and RegTx signals from the external MCU as well as the SWP and CWP. The CWP is updated on Jump-to-Subroutine

(JSR) and Return (RET) Instructions (decoded by the Transport and Control Instruction Circuitry).

Further details of all IOCU functions are presented in Chapter 5 together with circuit schematics and floorplans.

4.1.3. Instruction Set

The instruction set of the ARC processor is as shown in Table 4.1. The instruction set consists of 3 types of instructions: ALU, Transport, and Control. The format of the different types of instructions is shown in Fig. 4.6. To allow 2 instructions per 32-bit word, most instructions occupy 16 bits. To simplify decoding, instructions are composed of four 4-bit operand fields, F0-F3. As discussed in Chapter 3, packing 2 instructions per word is necessary to leave bandwidth on the IN bus for memory reads while executing one instruction per cycle.

4.1.3.1. ALU Instructions

ALU instructions are made up of three operand instructions, *Quick* instructions, two operand instructions, and SET instructions. The format of the different types of ALU instructions is shown in Fig. 4.6.

Three operand instructions are operations which access two inputs from the register file and store the result of the operation to the register file. The 4-bit F0 field selects the operation and is deemed the opcode. The inputs to the operation are indicated by F1 and F2 while F3 indicates the destination register for the result of the operation. As four codes of the F0 field are used to indicate longer opcode and *Quick* instructions, there can be only 12 three operand instructions. The selected set (see table 1) represents the most commonly used ALU operations.

Quick instructions (Add Quick (AQ) and Subtract Quick (SQ)) are implemented in ARC. These instructions allow adding or subtracting the contents of a register by

Table 4.1: ARC Instruction Opcodes

3 Operand ALU Instructions

Mneumonic	F0 (P0-P3)	F1 (P4-P7)	F2 F3	D6----D0
ADD	0000	S2	S1 DST	00x0 00x
AQ	0010	Q	S1 DST	00x0 10x
SUB	0001	S2	S1 DST	00x1 00x
SQ	0011	Q	S1 DST	00x1 10x
AWC	1000	S2	S1 DST	00x0 01x
SWC	1001	S2	S1 DST	00x1 01x
AND	0101	S2	S1 DST	0111 00x
OR	1011	S2	S1 DST	0111 01x
XOR	0111	S2	S1 DST	0111 10x
BIN	0110	S2	S1 DST	0110 11x
BEX	1010	S2	S1 DST	0110 10x

SET Instruction

SET Mneumonic	F0	F1=COND	F2 F3	D6----D0
=	0100	0010	S1/Dst S2	1001 001
=0	0100	0110	S1/Dst X	100x x11
<	0100	10x1	S1/Dst S2	1111 00x
<=	0100	00x1	S1/Dst S2	1101 00x
ODD	0100	11x1	S1/Dst X	111x x1x
LT	0100	0000	S1/Dst S2	1001 000
OV	0100	11x0	Dst X	101x x1x
!=	0100	10x0	S1/Dst S2	1011 00x

2 Operand ALU Instructions

Mneumonic	F0	F1	F2 F3	D6----D0
NOT	1110	11xx	S1 Dst	0111 11x
SL	1100	00xx	S1 Dst	0100 00x
SR	1101	00xx	S1 Dst	0101 00x
SRC	1101	01xx	S1 Dst	0101 01x
SLC	1100	01xx	S1 Dst	0100 01x
ROL	1100	10xx	S1 Dst	0100 1xx
ROR	1101	10xx	S1 Dst	0101 1xx
ROL8	1110	00xx	S1 Dst	0110 0xx

(I/O Control) Transport Instructions

Mneumonic	F0	F1	F2 F3
LD	1111	0001	S1 MMU
LDSR	1111	01xx	xx MMU
LDI	1111	1101	xx MMU
ST	1111	0011	S1 MMU

(I/O Control) Control Instructions

Mneumonic	F0	F1	F2 F3
JMP	1111	1011	S1 xxx
JSR	1111	1001	S1 xxx
BR	1111	1010	8 bit offset
RET	1111	1000	xx xxx
SKPONC	1111	0000	S1 xxx

3 Operand ALU Instructions

P0 F0 P3	F1	F2	F3
Op	S2@/Q	S1@	Dst@

2 Operand ALU Instructions

P0 F0 P3	P4 F1 P7	F2	F3
Op	Op	S1@	Dst@

Set Cond Instruction

P0 F0 P3	F1	F2	F3
0100	Cond	S1@,Dst@	S2@

2 Operand Transport Instructions

P0 F0 P3	P4 F1 P7	F2	F3
1111	Op	S1@	MMU@

Control Instructions

P0 F0 P3	P4 F1 P7	F2	F3
1111	BRANCH	Branch Offset	

P0 F0 P3	P4 F1 P7	F2	F3
1111	SKPONC	X	X

P0 F0 P3	P4 F1 P7		F3
1111	JSR/JMP /RET	S1	X

Fig. 4.6: Instruction Formats

as much as 15. This is most useful for loops and array indices. Quick instructions are operations which access one input from the register file, one input from the F1 field, and store the result of the operation to the register file. The F2 field selects the input from the register file while the F3 field indicates the destination register for the result of the operation. The F1 field of the instruction is known as the "Quick" operand because it is faster than fetching a constant from memory.

Two operand ALU instructions are operations which access one input from the register file and store the result of the operation to the register file. The input to the operation is indicated by F2 while F3 indicates the destination register for the result of the operation. F0 indicates that this is a two operand instruction while F1 specifies the operation. Two operand ALU instructions include unary logic operations (such as inversion, shifts, and rotates).

The SET instruction performs an operation on one or two inputs from the register file and writes the boolean result into bit 31 of a register. The F0 field of the SET instruction is 0100 (the SET opcode). The F1 field of the SET instruction is known as the COND field and indicates the condition to test. In addition, the SET instruction requires three operands: S1, DST, and S2 but only two fields are left. Therefore, a compromise must be made so the DST operand and the S1 operand occupy the same field: F2. The S2 operand occupies the F3 field, normally assigned to the DST operand. The SET instruction only returns one valid bit which is stored in bit 31 of the DST register. The other 31 bits contain the result of the ALU operation required by the SET instruction (usually subtraction). [3] SET can test for several conditions including overflow, S1 < S2(signed or unsigned), S1 <= S2, S1 odd, S1 = 0, S1 = S2, S1 != S2. The conditions are determined by the 4-bit COND field.

---

[3] With a minor modification to the ALU bitslice circuitry, it would be possible to leave bits 0-30 of SET's DST register intact. This is something that should be done in the next version of ARC.

The format of each of the instruction types shown in Fig. 4.6 and described above has been chosen to make the ALU instructions as orthogonal as possible. Each instruction has 4-bit S1 and DST register operands. 3 Operand ALU instructions also have an S2 address or Quick value occupying the F1 field of the instruction. The SET instruction compromises orthogonality to keep the instruction length to 16 bits.

In terms of function, the ALU instructions are straightforward and similar to those for other word-addressed RISC designs. Logic instructions include AND, OR, XOR, and NOT. Addition and subtraction can be done with carry or with a 4-bit "quick" operand. Rotation and Shifting can be done in either direction by one bit or by 8 bits left, or left with a carry (to enable multiplication). Character manipulation instructions include BIN and BEX which cause the least Significant byte of a word to be overwritten, or the least significant byte of a word to be isolated with zeros in all other bytes. The above byte or character manipulation instructions are essential because ARC is word-addressed. The SET instruction is very similar to a corresponding instruction in the MIPS machine[Henn82].

ALU instructions have few pipeline restrictions (see Section 4.1.4). They may be executed as either the first or second halfword instruction and may immediately follow branch or data transfer instructions as long as there are no data dependencies. Thus, the compiler commonly has the freedom to pack ALU instructions two per word and to reorder ALU instructions around branches. During the execution phase (PhiB) of an ALU instruction, the S1 and S2 busses become valid. On the following phase (PhiA) the DST bus becomes valid and the result of the ALU instruction is written to the register file. ALU instructions do not utilize the MMU bus, the IN bus, or the OUT bus. For more information about instruction timing and pipelining, see Section 4.1.4.

4.1.3.2. Transport Instructions

Transport instructions transfer data to/from the register file via the IN/OUT bus from/to external memory. The transport instructions include LoaD (LD), STore (ST), LoaD Special Register (LDSR), and LoaD Immediate (LDI). [4] The LD and ST instructions move data between a register (MMU bus operand) and external memory (address held in S1 register operand). External memory addresses are held in the register referenced by the S1 operand, and incoming or outgoing data is to be held in a register referenced by the MMU operand. The LDSR instruction can be used to retrieve 1 of 4 special registers from the external Memory Control Unit. These special registers include a PC, register stack or data stack pointers, etc. The LDSR instruction could easily have been split into 2 instructions such as LDSR and STSR. If the F2 field is to be used to identify an external special register, 16 special registers could be accessed.

Pipeline restrictions exist for transport instructions. Transport instructions can occupy only the first halfword instruction. Also, background register saves/restores cannot occur during the data phase of a transport instruction because transport instructions utilize the MMU bus of the register file as well as the IN or Out bus during that phase.

4.1.3.3. Control Instructions

Control Instructions perform data dependent alteration of control flow. Control instructions include JuMP (to a 32-bit address), BRanch (relative branch of +/-127), JSR and RET (subroutine jumps and returns change the active register window), and SKPONC (conditional skip of instruction). All branches and jumps are "delayed" so that the instruction word immediately following is executed in the same "context" (register window). The JSR and RET instructions are needed so that register window shifting as well as pushes and pops can work in conjunction with procedure calls and returns. The JuMP instruction has one operand, S1, a register address. The contents of that register will be sent to the OUT bus to become the new PC. The BRanch instruction contains an 8-bit offset which is sent through the OUT bus to be added to the current PC offchip. SKPONC has one operand, S1, a register address. If bit 31 of that register is a 1, the instruction immediately following SKPONC will have no effect (ALU result not stored in register file and Transport/Control instruction cancelled). [5] To perform a conditional branch, jump or subroutine call, a BR, JMP, or JSR instruction is placed immediately after a SKPONC instruction. To perform an efficient conditional assignment or conditional execution of an ALU instruction, an ALU instruction is placed immediately after a SKPONC instruction. This differs from the conditional branch instruction of the Stanford MIPS [Henn82].

---

[4] A LoaD Immediate (LDI) instruction is also necessary for simpler compilation and initialization but this instruction has some costs associated with it. This instruction is not orthogonal and it involves interpreting the following 32-bit instruction word as data and executing 2 NOPs. In the version of ARC implemented, a LDSR instruction was included but the LDI instruction was not fully implemented.

Pipeline restrictions exist for control instructions. Control instructions may always occupy the first halfword instruction. As well, SKPONC, BR and JuMP control instructions may occupy the second halfword instruction slot if they do not follow a STore instruction. Background register saves (pushes) may not occur during the execution phase of second-halfword BR and JuMP control instructions. The JSR, BR and JuMP instructions utilize the OUT bus during execution while SKPONC does not. The JSR instruction is set to always occupy the first halfword instruction to minimize pipeline conflicts and to minimize complexity in synchronizing the instruction fetches and incrementing the CWP.

The transport and control instructions send a corresponding 4-bit I/O control instruction (Mx0-3) to the memory control unit when they enter the decode pipestage (see Section 5.3). The Mx0-3 code notifies the External Memory Controller of the current Transport or Control instruction, or the push or pop status. This allows the MCU to match the pipeline of ARC and service the memory requirements of Transport and Control instructions as well as background register saves/restores. Alternatively, it is possible for the memory control unit to decode the Transport and Control instructions before sending them to the ARC. In this case, the 4-bit I/O control instruction to the MCU would be redundant.

---

[5] The SET COND ALU instruction usually works in conjunction with the SKPONC instruction.

4.1.4. Pipeline and Resource Scheduling

The purpose of pipelining is to increase the clock speed and to increase the utilization of the processor's resources. The pipeline of ARC instructions vary from three to four pipestages. [6] Pipelining is further complicated by the concept of *even* and *odd* clock phases, limited resources, and unidirectional busses. The pipestages of the ARC pipeline are shown in Fig. 4.7(a) and are explained as follows:

IF

> During the Instruction Fetch pipestage, an instruction word is read into the ARC via the IN bus and separated into two instructions. The Instruction Fetch pipestage always occurs on an even phase because the ARC processor fetches two instructions per 32-bit word every second phase.

Decode

> During the Decode pipestage, an instruction is separated into an opcode and operands. The opcode is converted into control signals by the ALU or Control decoder and the operands are interpreted according to the instruction type. The ALU and Control decoders queue control signals to the ALU and I/O Control logic while operand addresses are queued for the register file decoder. It is to be noted that instructions which occupy the first halfword of the instruction pair are decoded first on the odd phase immediately following the fetch, while second halfword instructions are

---

[6] The pipeline of load instructions may be extended beyond five pipestages if ARC is interfaced to slow memory or a cache miss occurs (see Section 3.4, pipelined memory access).

Pipeline of First Halfword Instruction

Phi 0	Phi 1		
I F	Decode	S1 Addr Execute BR Addr	Dst/MMU Data

Pipeline of Second Halfword Instruction

I F		Decode	Execute	Dst Data

Fig. 4.7 a) Pipeline and Resource Scheduling

	IN BUS	OUT BUS
Phi1 (Odd Phase)	LD Data Pop	ST Data Push Data BR/JMP Addr
Phi0 (Even Phase)	Instruction Fetch	LD/ST Addr JSR Addr BR/JMP Addr Push Data

Fig. 4.7 b): Resource Scheduling

decoded during the following even phase. (See top of Fig. 4.7(a))

Execute

Behaviour during the Execute pipestage depends on the instruction being performed. For ALU instructions, the Register File writes the register operands onto internal busses (S1 and ¯S2 busses) and the ALU executes the specified operation, queuing the result for writing to the register file on the following pipestage.

For most transport or control instructions, this pipestage is used to access a register which contains the external memory address to be written out the OUT bus. For example, during the execute pipestage of a LD instruction, the address of external memory to load from would be obtained from a register and written on the OUT bus (See Fig. 4.8).

The relative branch instruction writes the 8 bit branch offset on the OUT bus during this pipestage.

Data

The result of the ALU instruction is written to the register file via the Dst bus during the precharge phase of the Data pipestage. This increases performance by allowing more time to execute the operation (which is in ARC's critical path) while still allowing the data to be used as an argument for the following instruction.

For transport and control instructions, this pipestage is used for data transfer between the register file and external memory. This data transfer occurs via the MMU bus and the IN or OUT bus for incoming or outgoing data respectively. For the LoaD instruction, this pipestage may be extended by an even number of clock phases if the processor is interfaced to slow RAM or a cache miss occurs (see Sections 3.4 and 5.3.3 for more

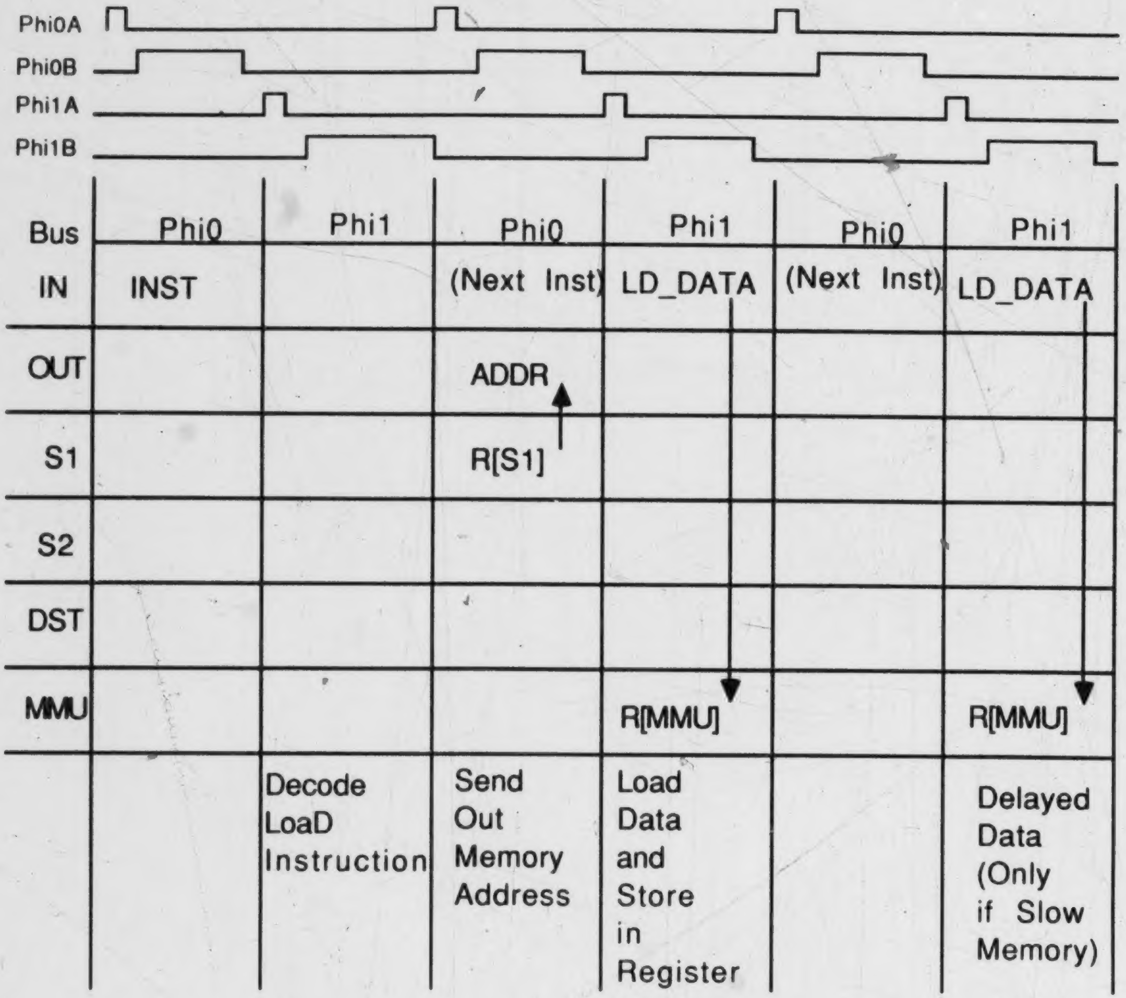

Phi0A
Phi0B
Phi1A
Phi1B
Bus
Phi0
Phi1
Phi0
Phi1
Phi0
Phi1
IN
INST
(Next Inst)
LD_DATA
(Next Inst)
LD_DATA
OUT
ADDR
S1
R[S1]
S2
DST
MMU
R[MMU]
R[MMU]
Decode LoaD Instruction
Send Out Memory Address
Load Data and Store in Register
Delayed Data (Only if Slow Memory)

Fig. 4.8: Timing of LoaD Instruction

information).

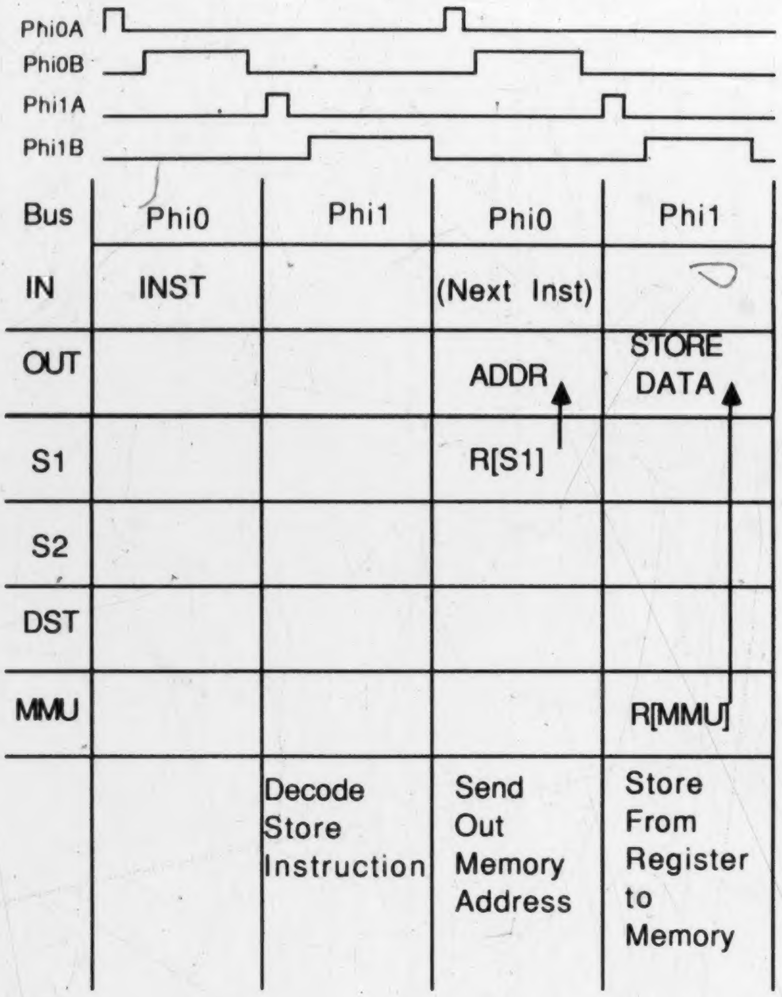
Examples of the pipeline timing of LoaD, STore and ALU instructions are given in Figs. 4.8, to 4.11.

Figure 4.8 depicts the timing and bus resource utilization of the LD instruction. The LD instruction is composed of 4 pipestages: fetch, decode, address transfer, and data transfer. The address transfer pipestage utilizes the S1 bus and the OUT bus to transfer the address from a register to the MCU. The data transfer pipestage utilizes the IN bus and the MMU bus to receive incoming data (from the above memory address) and store it in a register via the MMU bus.

Fig. 4.9 depicts the timing and bus resource utilization of the ST instruction The ST instruction is composed of 4 pipestages: fetch, decode, address transfer, and data transfer. The address transfer pipestage sends the address over the S1 bus to the OUT bus. The data transfer pipestage sends the data to be stored in memory over the MMU bus to the OUT bus.

Figures. 4.10 and 4.11 depict the timing and bus resource utilization of first or second halfword ALU instructions repectively. Instructions which occupy the second halfword, decode, execute, etc. one phase later than the instruction occupying the first halfword. During the decode phase, no busses are utilized. During the execution phase, ALU instructions utilize the S1, ~S2, and DST busses. Following the Execution Phase, the result of the ALU instruction is stored into the register file. This actually occurs during the precharge phase of the following pipestage.

Fig. 4.7(b) shows how the IN and OUT bus resources are scheduled. IN and OUT bus scheduling has been organized to minimize the pipeline length of instructions, to minimize conflicts between instructions and to balance the I/O bus bandwidth between the two halfwords. In addition to scheduling the IN and OUT busses, it is necessary to ensure that conflicts do not occur on internal busses (S1, ~S2, Dst, MMU



Phi0A
Phi0B
Phi1A
Phi1B

Bus	Phi0	Phi1	Phi0	Phi1
IN	INST		(Next Inst)	
OUT			ADDR	STORE DATA
S1			R[S1]	
S2				
DST				
MMU				R[MMU]
		Decode Store Instruction	Send Out Memory Address	Store From Register to Memory

Fig. 4.9: Timing of STore Instruction

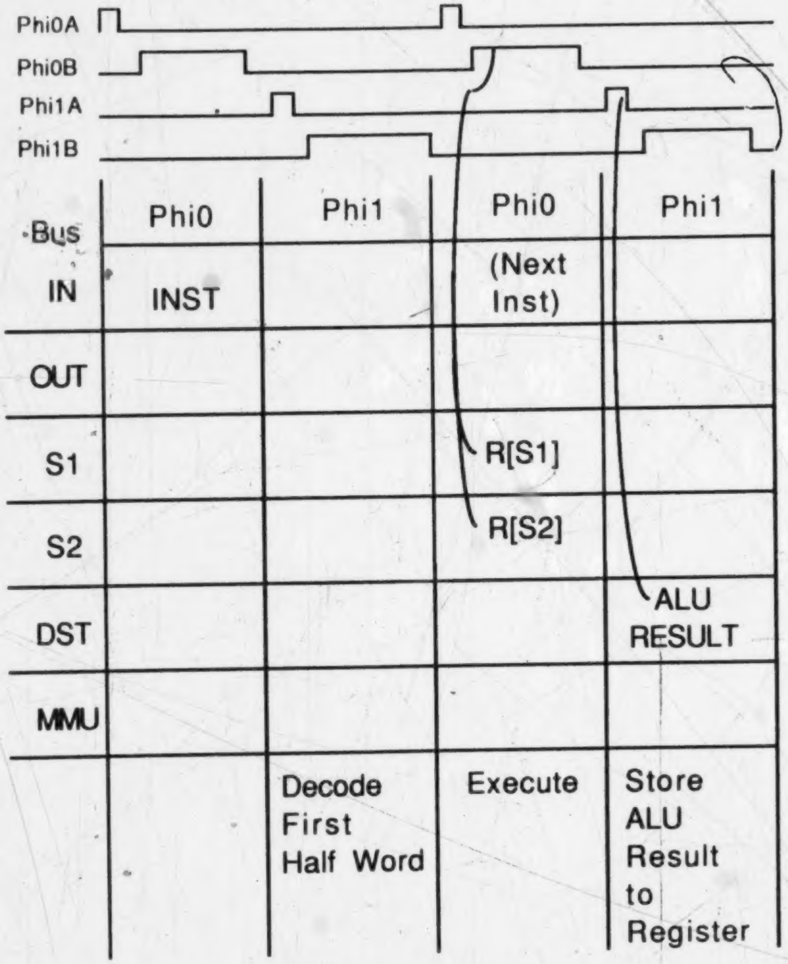

Phi0A
Phi0B
Phi1A
Phi1B
Bus
Phi0
Phi1
Phi0
Phi1
IN
INST
(Next Inst)
OUT
S1
R[S1]
S2
R[S2]
DST
ALU RESULT
MMU
Decode First Half Word
Execute
Store ALU Result to Register

Fig. 4.10: Timing of First-Halfword ALU Instruction

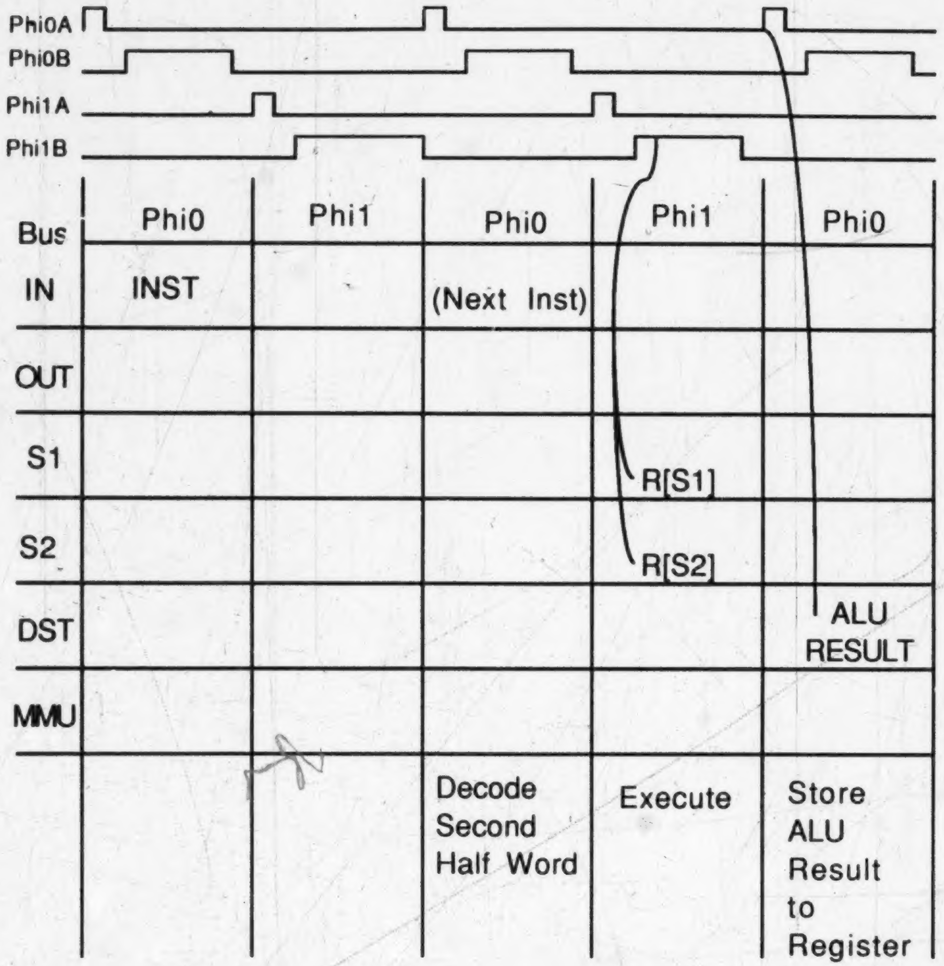

Phi0A
Phi0B
Phi1A
Phi1B
Bus
IN
OUT
S1
S2
DST
MMU
Phi0
Phi1
Phi0
Phi1
Phi0
INST
(Next Inst)
R[S1]
R[S2]
ALU RESULT
Decode Second Half Word
Execute
Store ALU Result to Register

Fig. 4.11: Timing of Second-Halfword ALU Instruction

busses). The example timing diagrams of Figs. 4.8-4.11 also show the bus utilization for the internal as well as the external busses for LoaD, STore, and ALU instructions.

The IN bus must handle incoming instructions, loaded data, and data restored to registers. Instruction pairs must come in every other phase (if we are to execute one instruction per cycle) so the obvious choice is to schedule LD data and restored register data in the same phase. To avoid conflict on the IN bus, background register restores (Pops) cannot occur while a LD instruction is utilizing the IN bus. Instruction fetches are arbitrarily set to occur during even clock phases.

The OUT bus handles BRanch/JuMP addresses, JSR addresses, STored data, LD/ST addresses, and background register saves ("push"es). To shorten the pipeline for LoaDs and STores, LD/ST addresses should be sent out on an even Phi after the decode, and STore data should be sent out on the following odd clock phase. For the sake of simplicity, JSR addresses should also be sent out on the even Phi after the decode. The above decisions create the restriction that JSRs, LoaDs and STores can only be first halfword instructions. BRanch/JuMP addresses and Pushes should be scheduled to shorten the pipeline, provide an even utilization of the OUT bus as well as, if possible, provide a balance of first halfword and second halfword instructions. With this criteria in mind, there are 2 possibilities:

1) Branch/Jump addresses could be scheduled during even chock phases and pushes could be scheduled during odd phases. Therefore, branches and jumps would be restricted to being first halfword instructions.

2) BRanch/JuMP instructions could alternatively occur in either phase for better balancing of first and second halfword instructions. This would, however, create a conflict between the BR/JMP addresses from second-halfword BR/JMP instructions and Pushed Data as well as ST Data.

The second possibility is more efficient because it provides better balancing of first and second halfword instructions. It is more costly because it requires slightly more hardware to arbitrate pushes and adds a compiler restriction.

The following table lists the halfword restrictions for the various instructions and other ARC operations.

Table 4.2: Instruction/Operation Restrictions

Operation/ Instruction	Halfword (Pipeline) Restrictions
ALU Instructions	Either First or Second Halfword
SET Instruction	Either First or Second Halfword
BR, JMP, SKPONC	Either First or Second Halfword
LD, LDI, LDSR	First Halfword Only
ST, STSR, JSR	First Halfword Only
PUSH	During ALU Instruction Which Does not Follow LD/ST/LDSR Only
POP	During First Halfword ALU Instruction Only

4.2. ARC System Architecture

ARC is significantly different than conventional processors in a number of ways, the most obvious of which are the unidirectional busses and the interface to the Memory Control Unit. These differences must be reflected in a system built around ARC. Several possible system configurations are shown in Figs. 4.12, 4.13, and 4.14. The major components of these systems include the Alberta Risc Chip(ARC) instruction processor, a Memory Control Unit (MCU) and a RAM subsystem. Fig. 4.12 portrays a system in which ARC and the MCU are connected to a single port RAM memory subsystem. Fig 4.13 features dual port RAM to increase the available I/O

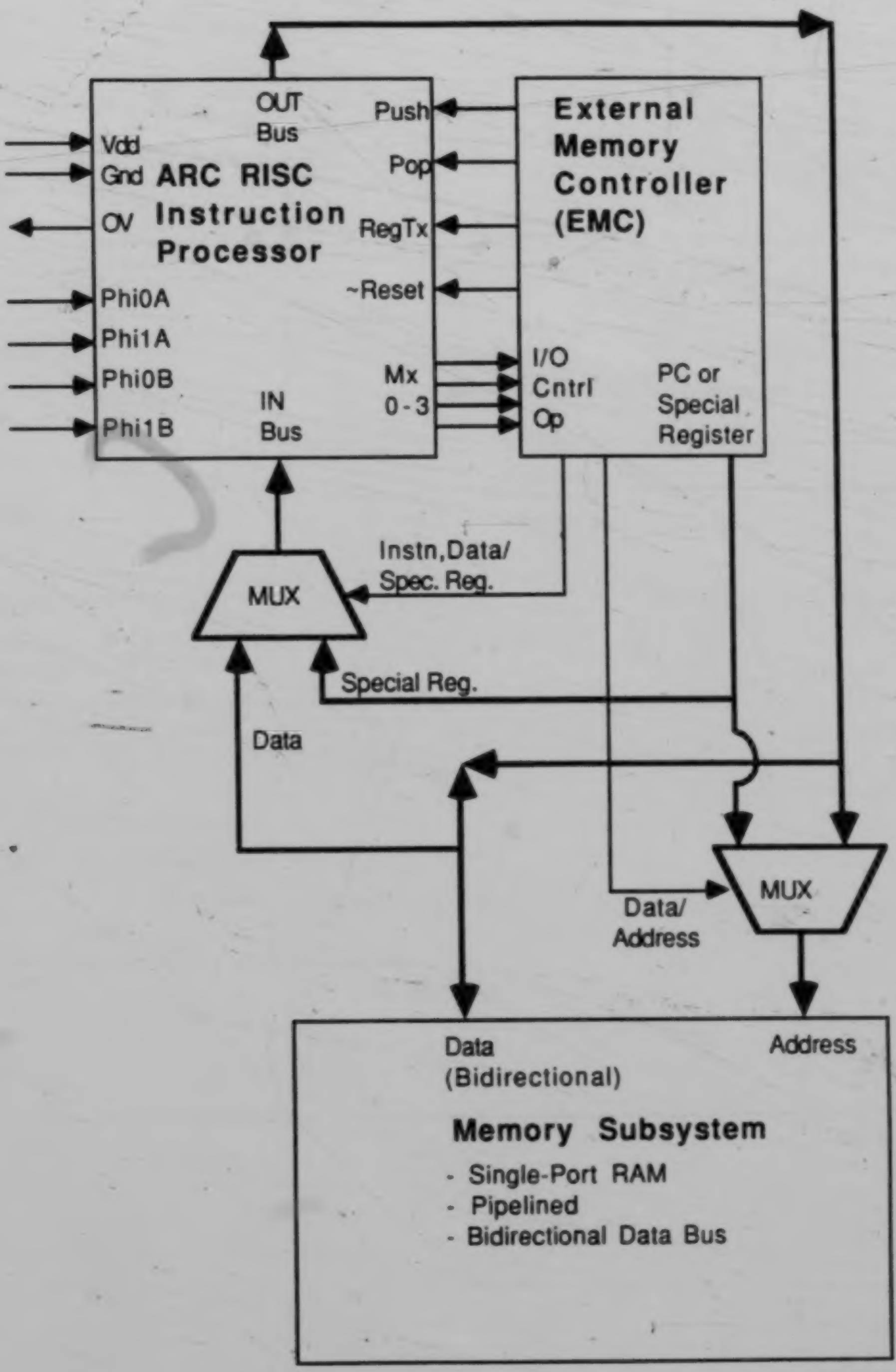

OUT
Bus
Vdd
Gnd
ARC RISC
Instruction
Processor
OV
Phi0A
Phi1A
Phi0B
Phi1B
IN
Bus
Push
Pop
RegTx
~Reset
Mx
0 - 3
External
Memory
Controller
(EMC)
I/O
Cntrl
Op
PC or
Special
Register
Instn,Data/
Spec. Reg.
MUX
Special Reg.
Data
MUX
Data/
Address
Data
(Bidirectional)
Address
Memory Subsystem
- Single-Port RAM
- Pipelined
- Bidirectional Data Bus

Fig. 4.12: ARC Interface to Single-Port RAM

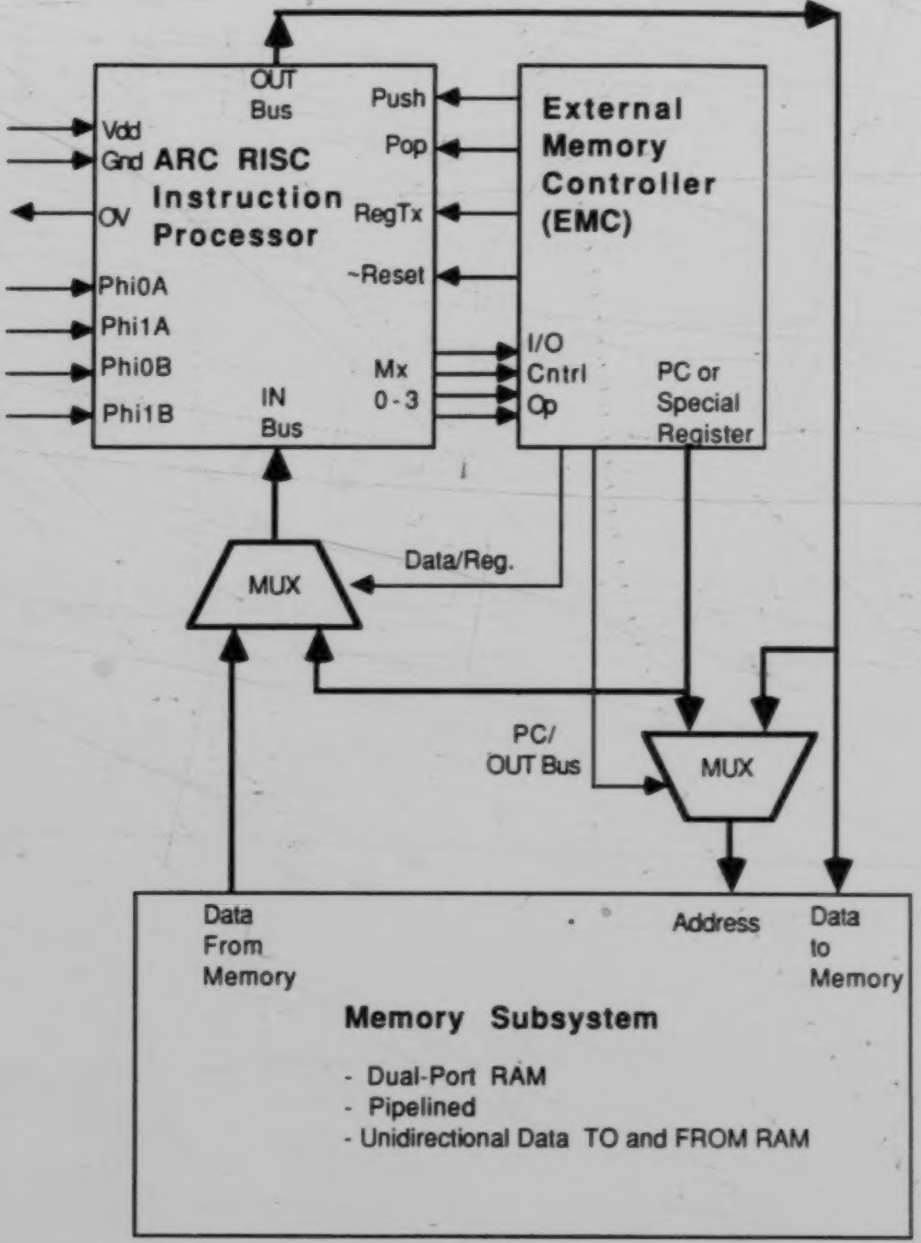

OUT Bus
Push
Pop
RegTx
~Reset
Vdd
Gnd
ARC RISC Instruction Processor
OV
Phi0A
Phi1A
Phi0B
Phi1B
IN Bus
Mx 0-3
External Memory Controller (EMC)
I/O Cntrl Op
PC or Special Register
Data/Reg.
MUX
PC/ OUT Bus
MUX
Data From Memory
Address
Data to Memory
Memory Subsystem
- Dual-Port RAM
- Pipelined
- Unidirectional Data TO and FROM RAM

Fig. 4.13: ARC Interface to Dual-Port RAM

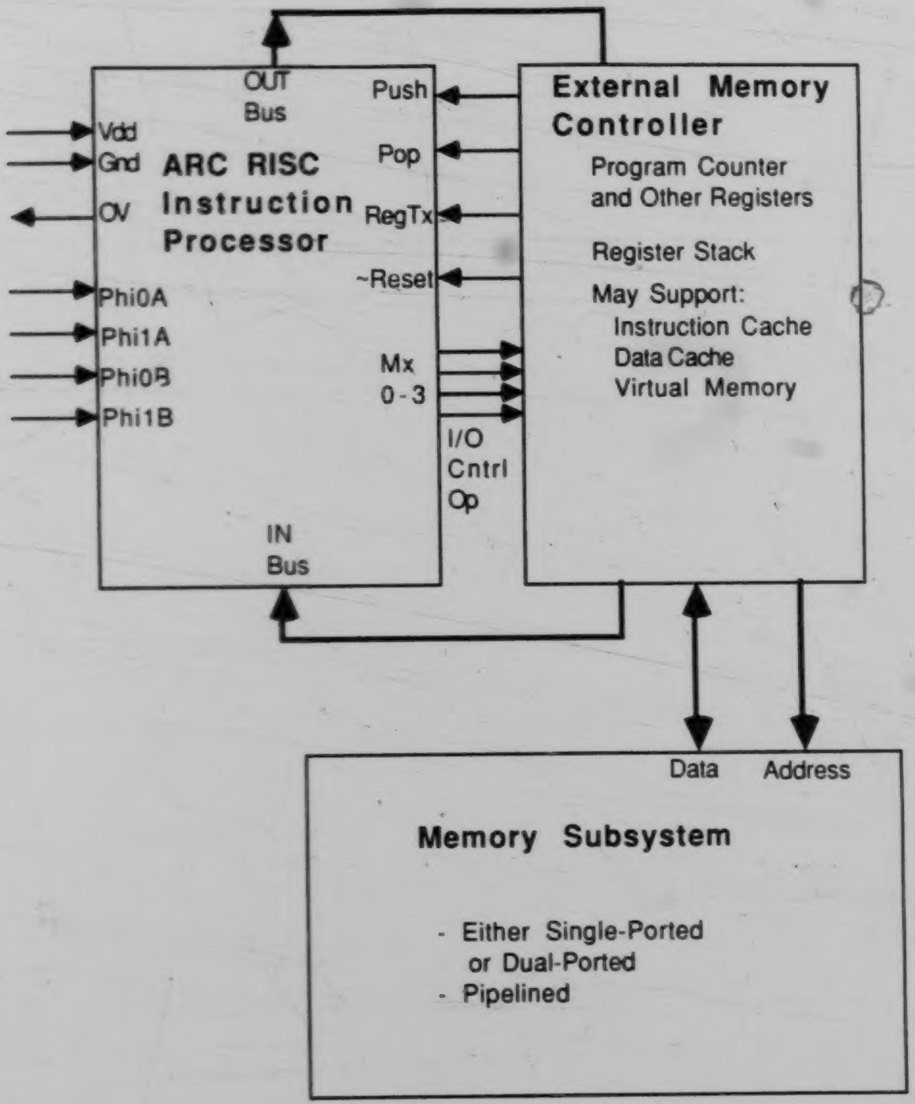

OUT
Bus
ARC RISC
Instruction
Processor
Vdd
Gnd
OV
Phi0A
Phi1A
Phi0B
Phi1B
Push
Pop
RegTx
~Reset
Mx
0 - 3
I/O
Cntrl
Op
IN
Bus
External Memory
Controller
Program Counter
and Other Registers
Register Stack
May Support:
Instruction Cache
Data Cache
Virtual Memory
Data
Address
Memory Subsystem
- Either Single-Ported
or Dual-Ported
- Pipelined

Fig. 4.14: ARC System Diagram

bandwidth. The block diagram of Fig 4.14 shows how a sophisticated MCU which includes virtual address translation, cache, security and interrupt handling could be interfaced to the memory subsystem and ARC. It is also possible that ARC and the MCU could be implemented on the same chip but this option is not investigated because it is currently beyond the capability of technologies available at our university. As we will see, the MCU must be tailored specifically for the system/memory architecture and may be considered as the "intelligent" part of the memory subsystem.

The choice of a memory subsystem and type of MCU depends on a number of factors. The simple system portrayed in Fig. 4.12 with single port RAM and a simple MCU is the cheapest and lowest performance system. In this system, the MCU logic is distributed within the RAM subsystem. The performance of this system would be limited by the clock speed and comparatively low memory bandwidth of the RAM subsystem. Thus, connecting ARC to conventional single port RAM defeats the I/O bandwidth gains of ARC's busses. The dual port RAM system of Fig. 4.13 offers greater memory bandwidth because dual port RAM provides 2 busses to match ARC's busses but the throughput of the system is still limited by the clock speed of the dual port RAM subsystem. The clock speed could be increased beyond the speed of the RAM if page mode interleaving or caching is incorporated into the system. Figure 4.14 portrays a higher performance system which would be able to achieve high I/O bandwidth and a high clock speed. In addition, this system shows that functions such as virtual address translation, etc. would be incorporated into the MCU. The MCU could support either unified or divided instruction and data caches. Separate data and instruction caches would result in a system architecture similar to the Harvard Architecture.

4.2.1. Interface of ARC to Memory Control Unit

The interface/division between ARC and the MCU was discussed in Chapter 3 at a philosophical level. Chapter 3 also introduced several novel features of ARC which are dependent upon MCU support. In the previous section, the example system configurations of Figs. 4.12, 4.13, and 4.14, show some details of how the MCU can be interfaced to ARC and the memory subsystem. This section analyzes the interface and functions necessary within the MCU to support the philosophical issues, the novel features of ARC as well as optional MCU functions which would enhance the system performance.

In the ARC, some functions which are normally incorporated as part of a microprocessor are off-chip. For example, special registers such as the Program Counter (PC) and Register Stack Pointer are implemented in the Memory Control Unit. PC manipulation and sequencing, stack management, special register management, I/O control, and instruction and data caching must therefore be performed off-chip. ARC is referred to as an *instruction processor* because it only processes instructions and depends on the MCU to "feed" it instructions. The MCU is responsible for functions which are not normally implemented in a MMU and has a sibling(parallel) rather than a master-slave relationship with the processor. Other functions to enhance system performance are also to be associated with the MCU rather than the CPU as is common in conventional architectures. These optional functions could include virtual memory translation, interrupt handling, and instruction preprocessing.

4.2.1.1. Program Counter and Special Registers of the MCU

Because instructions originate from memory and the generation of instruction addresses seldom requires CPU intervention, the PC and associated sequencing logic is implemented in the MCU. One advantage of moving the program counter (PC) off-chip is in the reduction of necessary bandwidth between the instruction processor and memory controller. Since the PC is not on-chip, it no longer is necessary to send out the PC address on each instruction fetch. Instruction sequencing is performed by adding 1 to the PC during sequential instruction execution. When a jump instruction occurs, the new PC is loaded from the ARC register file via the OUT bus. When a relative branch (BR) occurs, the 8-bit offset of the branch instruction is added (2's complement) to the current PC. The branch offset can be received from the OUT bus of ARC or can be extracted from the BR instruction as it is sent to ARC.

In addition to the PC, other special registers such as the Register Stack Pointer are also associated with the MCU because they address memory locations used by memory control functions. The special registers of the MCU also require additional logic such as an adder to perform incrementing, decrementing and offset addition.

4.2.1.2. The Register Stack of the MCU

The MCU must control the register stack which is associated with the circular window structure of the register file. When subroutine calls occur(JSR instruction), a "new" register window is allocated to the called subroutine. If several subroutine calls occur without returns, registers will be overwritten unless they are saved to the register stack of the MCU. Registers are retrieved from the register stack when returns from subroutines occur (RET instruction). The register stack is operated on primarily by the background register save/restore processes discussed in Chapters 3, 4 and 5. The register stack structure is depicted in Fig. 4.4 and introduced in Section 4.1.1.2.

4.2.1.3. MCU I/O Control

The I/O requirements of the IN and OUT busses of ARC can be controlled/served by the MCU. The instruction requirements of ARC can be served because of the simplicity of the instruction set (only 1 addressing mode, simple control flow). Other I/O requirements such as data and addresses for LoaDs or STores, background register saves/restores, and special register transfers, are facilitated by the interface between ARC and the memory controller.

The fast, efficient, low-pin-count unidirectional-bus interface, which is defined between ARC and the memory controller facilitates access to memory, special registers, and other MCU services. The interface between ARC and the MCU varies somewhat between the systems presented in Section 4.2. In general, communication is via ARC's IN and OUT busses, the 4-bit I/O control opcode, Mx0-3, and via the RegTx, Push, and Pop signals. The Mx0-3 code indicates the instruction currently being decoded by ARC. Based on this, the MCU knows what ARC will be sending on the OUT bus and what ARC will be requiring on the IN bus in subsequent cycles. It is then the responsibility of the MCU to redirect the data and addresses from the OUT bus to the memory subsystem and to direct data and instructions from the memory subsystem to the IN bus. The MCU must therefore be pipelined to match ARC so that information is expected and dispatched during the correct cycle. The Push, Pop, and RegTx signals originating from the MCU direct the *background* register save and restore functions. The RegTx signal occurs when it is necessary to either save or restore registers. If the Push and RegTx signals are high, registers will be saved from ARC to the memory subsystem (or MCU) and the next window of ARC's register file becomes the current window. If the Pop and RegTx signals are high, registers will be restored from the memory subsystem back to their original register subwindow of the ARC Register File and the previous window of ARC's register file becomes the current window. It is possible for either the push or pop signals to be asserted without a register transfer being required. When this happens, ARC simply shifts to the next/previous register window without transferring registers. For more information about the above I/O control mechanisms, see Chapter 5.

4.2.1.4. MCU Caching

An instruction and or data cache can be incorporated into the MCU to obtain the same benefits as in a traditional RISC system. The interface between ARC and the MCU is however unique so that the cache implementation may be different than in a traditional RISC.

As detailed in Chapter 2, a variety of caching schemes exist in current RISC architectures. The Motorola 88000 RISC processor follows the Harvard Architecture which involves separate data and instruction caches and separate instruction and data busses while Intel processors maintain a unified instruction and data caching scheme [Furl89]. In many conventional RISCs, the processor is designed with a certain caching scheme in mind.

The ARC chip can be interfaced to a variety of MCU cache architectures depending upon the memory architecture and bandwidth required. For example, the instruction and data caches could be separated or unified and the MCU can be connected to single- or dual-port RAM. If the instruction and data caches are separate, it would be wise to couple the program counter with the instruction cache control and the register stack with the data cache control. Dual-port RAM would provide more bandwidth between the MCU (and therefore the cache) and the external RAM although if the cache has a high enough hit ratio, single port RAM may be sufficient. With dual-port RAM, it would probably be best to have unidirectional read and write busses to the RAM but instruction and data busses as in the Harvard Architecture are also possible between the MCU and RAM.

4.2.1.5. Optional MCU Functions to Enhance Performance

Virtual memory translation, interrupt handling, and instruction preprocessing are features which, if implemented, are normally incorporated into a CPU or CPU coprocessor. In the ARC system these functions would be associated more closely with the MCU. Virtual memory translation can either preceed or succeed the cache. Interrupt handling is also possible and is discussed along with the philosophical issues near the end of Chapter 3. Instruction preprocessing is also a possibility because the MCU could decode and preprocess some instructions before sending them to ARC.

5. ARC CIRCUIT/HARDWARE DESIGN DESCRIPTION

As mentioned previously, The ARC is composed of 3 main parts, The ALU, Register File, and the I/O Control Unit (IOCU). This chapter gives details of the hardware design of each of these parts at the floorplan and schematic levels. Sufficient information will be presented here to understand the design without getting into process-dependent design details. (Device level layouts and simulations are presented in a separate technical report.) As well, given the floorplan and schematic information presented in this chapter, an IC designer should be able to inspect functional blocks in the IC layout and to more fully understand the implementation details of ARC's novel features, presented in Chapter 3.

5.1. The Arithmetic and Logic Unit

The floorplan of the ARC ALU is depicted in Fig. 5.1.1. The ALU is composed of 5 sections: an adder, a decoder, bitslice circuitry, a condition tester, and other non-bitslice circuitry. Much of the ALU circuitry is MUX-based which made it more regular and compact than purely combinational circuitry. The ALU is connected to the register file through 3 unidirectional busses, S1, ˜S2, and DST. The S1 and ˜S2 busses carry the contents of 2 registers to the ALU and the DST bus returns the result of an ALU operation to a register. The ALU instruction byte (P0-P7) is passed to the ALU decoder which generates internal ALU control signals (D0-D6). An overflow signal (OV) is available from the ALU. Two clock signals, PhiA and PhiB, synchronize the ALU with the rest of ARC.

According to simulations, the time required for the ALU execution phase (PhiB) will set the speed of the processor. During PhiB, the register file conditionally discharges the S1 and ˜S2 busses and the ALU performs the specified operation. The critical path within the ALU in from the S1 and ˜S2 busses, through the adder and the condition/flags section to the DST bus to implement the SET instruction. Therefore,

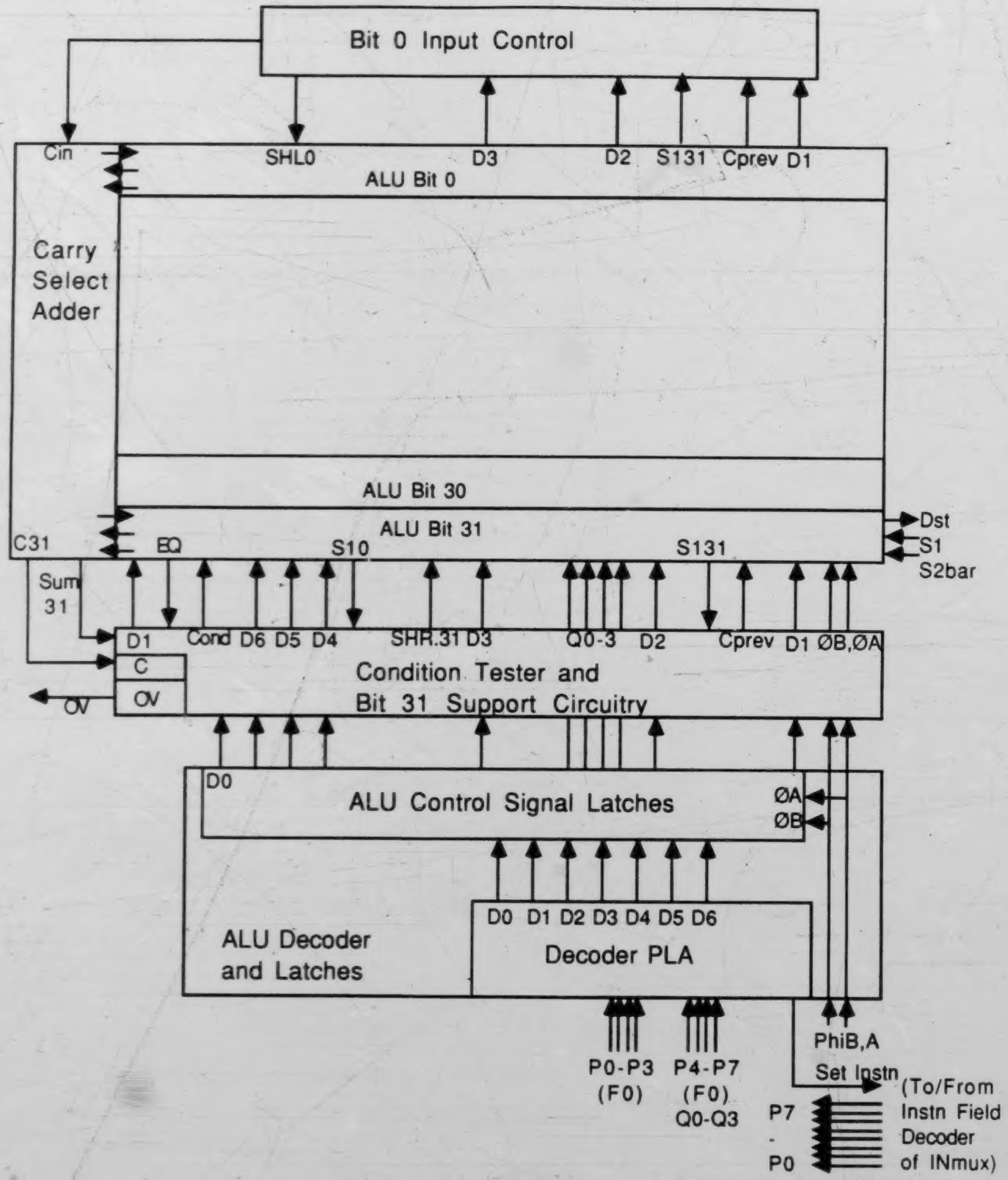

Bit 0 Input Control
Cin
SHL0
D3
D2
S131
Cprev
D1
ALU Bit 0
Carry
Select
Adder
ALU Bit 30
ALU Bit 31
Dst
S1
S2bar
C31
EQ
S10
S131
Sum
31
D1
Cond
D6
D5
D4
SHR.31
D3
Q0-3
D2
Cprev
D1
ØB,ØA
C
OV
OV
Condition Tester and
Bit 31 Support Circuitry
D0
ALU Control Signal Latches
ØA
ØB
D0
D1
D2
D3
D4
D5
D6
ALU Decoder
and Latches
Decoder PLA
P0-P3
(F0)
P4-P7
(F0)
Q0-Q3
PhiB,A
Set Instn
(To/From
Instn Field
Decoder
of INmux)
P7
-
P0

Fig. 5.1.1: ALU Floorplan

these sections were optimized for speed. The logic circuitry which implements other opcodes was optimized for area.

The design of an ALU for the ARC was first tackled by Tai An Ly during the fall of 1986. Since then the ALU has undergone some functional changes. The adder's layout was redone by Alan Mitchell during the fall of 1987. The rest of the ALU was redesigned and layout redone as part of this thesis during 1988.

5.1.1. ALU Adder

The adder is a 32-bit carry-select adder very similar to the adder design presented in page 331 of the text, "Principles of CMOS VLSI Design" by Weste and Eshraghian [Weste85]. Bitslice I/O signals to/from the adder include the *a*, *b*, and *sum* signals. Nonbitslice I/O signals include the *carry-in* and the *carry-out*. The *a* and *b* signals are the two 32-bit operands to undergo 2's complement addition. The *sum* signal is the 32-bit result of that addition. More information about the design of this adder is presented in [Ly87] and [Mitc87].

5.1.2. ALU Decoder

The ALU instruction decoder translates the instruction opcodes (P0-P7) into the ALU control signals (D0-D6) and holds them valid throughout the execution phase of the ALU. The ALU decoder consists of a PLA to convert opcodes into ALU control signals and latches to hold the control signals. Table 4.1 lists all ALU instructions together with corresponding instruction opcodes (P0-P7) and ALU control signals (D0-D6).

The bottom of Fig. 5.1.1 displays significant features of the ALU decoder and associated latches. The decoder PLA converts the instruction opcode (P0-P7) inputs into the ALU control signals (D0-D6) and 3 other signals (Set, 3op, 2op) which indicate the current instruction type (Set condition, 3 operand, or 2 operand instruction).

The *Set* signal is required by the inmux of the IOCU (see section 5.3) and the *3op* and *2op* signals are used to simplify the PLA logic within the ALU decoder. The *Q0-Q3* signals are electrically the same as P4-P7 and are used as a 4-bit *Quick* (immediate mode) constant for the AddQ and SubQ instructions.

The ALU decoder PLA operates during PhiB of the decode pipestage. At the end of this PhiB, *D0-D6* and *Q0-Q3* are latched to hold these signals valid according to the pipeline timing diagrams for ALU instructions given in Fig. 4.11 and 4.12. [1] During the following PhiA (precharge) phase, the signals are propagated to the next latch and begin to drive the ALU signal lines. The *D0-D6* and *Q0-Q3* signals thus remain valid throughout the following ALU execution phase (PhiB) until the following PhiA.

The ALU decoder PLA equations are given in Fig. 5.1.2(b). These equations were derived from the opcodes in Table 4.1 and the ALU decoder table of Fig 5.1.2(a).

5.1.3. ALU Bitslice Circuitry

The ALU bitslice is the heart of the ALU and implements all of the logic operations. The ALU contains 32 bitslices stacked with bit0 at the top and bit31 at the bottom (nearest the decoder). The ALU control signals (D0-D6) from the ALU decoder control the flow of operands through the ALU bitslice multiplexers and thereby control which operation is performed.

Floorplan diagrams of the ALU bitslice circuitry are given in Fig. 5.1.3. Fig. 5.1.3(a) is the floorplan of a generic ALU bitslice which indicates the location of signals and functional blocks. The ALU bitslice circuitry is interfaced to the Register File at its right via the S1, $\overline{S2}$, and DST busses. The S1 bus carries the S1 operand from the register file to the ALU. The $\overline{S2}$ bus carries the inverse of the S2 operand

---

[1] Pipelining is discussed in detail in section 4.1.4 of this thesis.

6

ALU Signal	3 Operand Instruction	2 Operand Instruction (P0 * P1)	Set Cond Instruction (P0'P1P2'P3')	MMU Instruction (P0P1P2P3)
D0	X	X	P6	X
D1	P0	P5	P5	X
D2	P2P3' + P0'P2	P4	0	X
D3	P3	P3 except NOT	1	X
D4	1	P2	P4	X
D5	P1 + P0*P2	P1+ (P0*P2)	P7	X
D6	0	0	1	X

Fig. 5.1.2 a): ALU Decoder Table

ALU DECODER PLA EQUATIONS

3op = P1' + P0' * P2 + P0' * P3
2op = P0 * P1
SET Cond = P0' * P1 * P2' * P3'

D0 = P6
D1 = 3op * P0 + P5*2op + P5 * P0'P1P2'P3'
D2 = 3op * P2P3' + 3op * P0'P2 + 2op * P4
D3 = P0'P1P2'P3' + P3 + P0*P1*P4*P5
D4 = 3op + P2 * 2op + P4*P0'P1P2'P3'
D5 = P1 + P0P2 + P0'P1P2'P3''*P7
D6 = P0'P1P2'P3'

Fig. 5.1.2 b): ALU Decoder PLA Equations

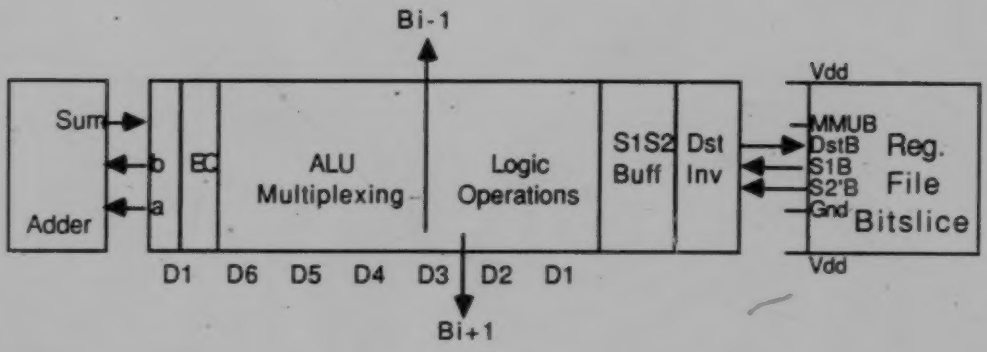

Bi-1
Sum
b
a
Adder
EC
ALU Multiplexing
Logic Operations
S1S2 Buff
Dst Inv
Vdd
MMUB
DstB
S1B
S2'B
Gnd
Reg. File Bitslice
Vdd
D1
D6
D5
D4
D3
D2
D1
Bi+1

Fig. 5.1.3 a): ALU Generic Bitslice Floorplan

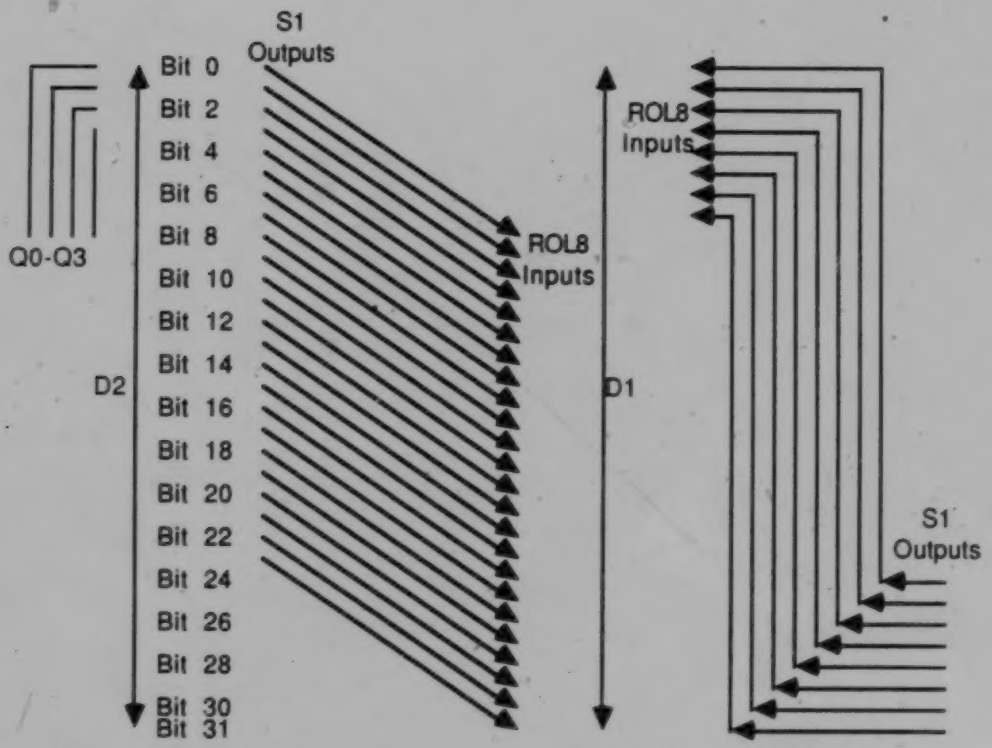

S1 Outputs
Bit 0
Bit 2
Bit 4
Bit 6
Bit 8
Bit 10
Bit 12
Bit 14
Bit 16
Bit 18
Bit 20
Bit 22
Bit 24
Bit 26
Bit 28
Bit 30
Bit 31
Q0-Q3
D2
ROL8 Inputs
ROL8 Inputs
D1
S1 Outputs

Fig. 5.1.3 b): Routing of ROL-8 Signals

from the register file to the ALU. The DST bus carries the result of an ALU operation to the register file. The Adder is interfaced to the left of the ALU bitslice via the *a*, *b*, and *Sum* signals. Each ALU bitslice is connected to other bitslices to enable shift and rotate operations. Single-bit shift and rotate operations are enabled by the *Bi-1* and *Bi*+1 signals shown in Fig. 5.1.3(a). Rotate-by-8 operations are facilitated by the inter-bitslice routing displayed in Fig 5.1.3(b). Fig 5.1.3(b) also indicates how the *Q0-Q3* signals are routed.

5.1.3.1. ALU Bitslice Schematics

The ALU Bitslice schematic is presented in Fig. 5.1.4. The ALU operates by computing all logic functions in parallel and then selecting the desired result with a mux tree built from 2-1 inverting muxes (see Fig. 5.1.6(a)). Minor differences that exist between the ALU bitslices are highlighted on the schematics.

The mux tree was designed to minimize the complexity of decoding the opcodes and to maintain performance. The logic operations (AND, OR, XOR, NOT) are implemented at the bottom right of the schematic and are selected by the *D1* and *D2* control signals. The desired Shift and Rotate functions are selected with the D3, and D4 control signals. The various Add and subtract functions are implemented by conditioning the b input to the adder as shown at the top of the schematic. (It is to be noted that the adder is connected directly to the output stage because the adder is in the critical path of the ALU bitslice.) The output stage of the ALU bitslice circuitry is composed of a pass transistor preceding an inverter. The inverter, in turn, drives the DST Bus. This pass transistor and inverter form a dynamic latch which is loaded during the ALU execution phase (PhiB). The "EQ Line" circuitry detailed above the output stage of the bitslice schematic of Fig. 5.1.4 tests for a zero sum or S1 operand. A more detailed schematic of the EQ Line Circuitry is given in Fig. 5.1.6(c).

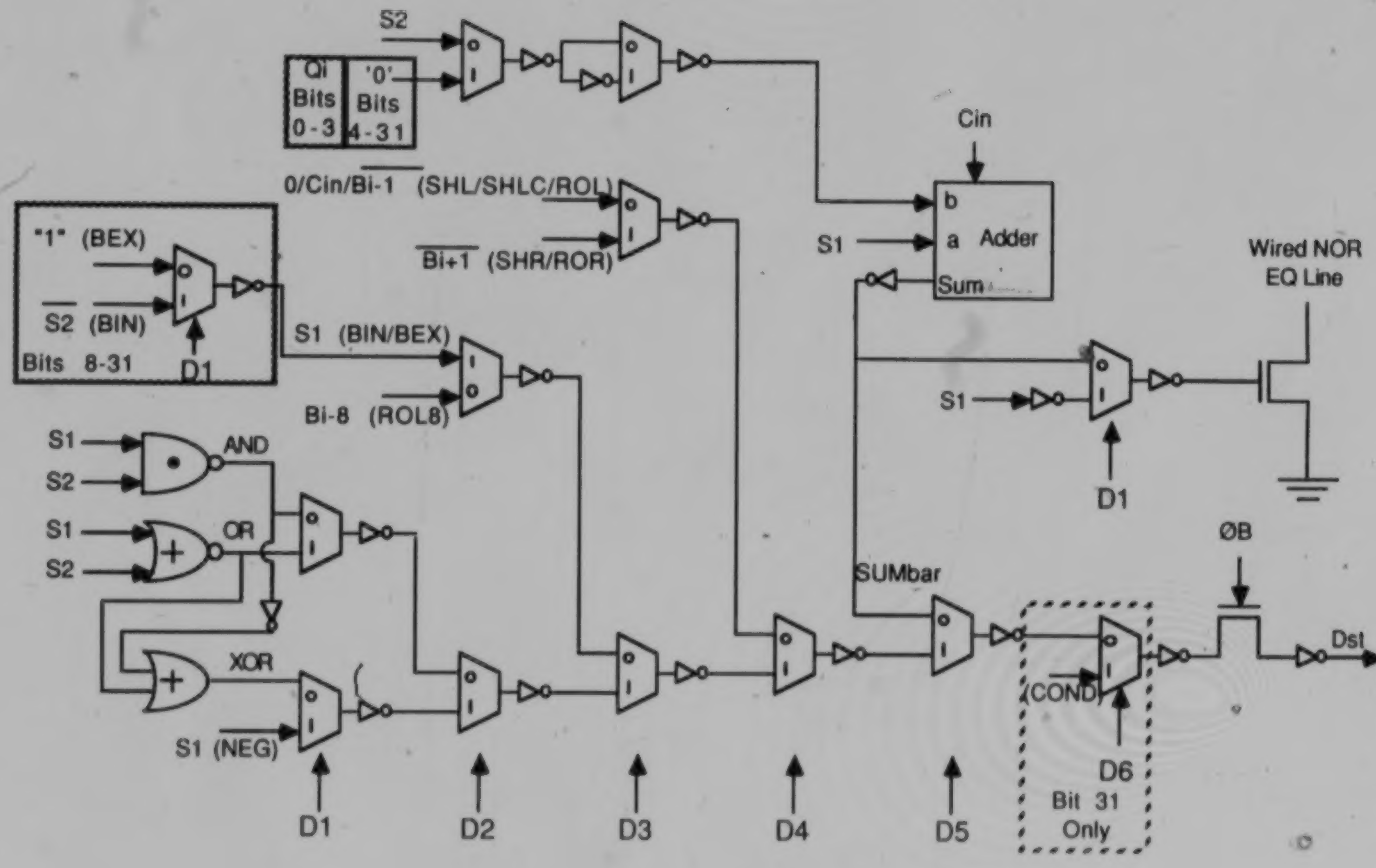

S2
Qi
Bits
0-3
'0'
Bits
4-31
Cin
b
a
Adder
Sum
S1
0/Cin/Bi-1 (SHL/SHLC/ROL)
Bi+1 (SHR/ROR)
"1" (BEX)
S2 (BIN)
Bits 8-31
D1
S1 (BIN/BEX)
Bi-8 (ROL8)
Wired NOR
EQ Line
S1
D1
S1
S2
AND
S1
S2
OR
XOR
S1 (NEG)
SUMbar
ØB
Dst
(COND)
D6
Bit 31
Only
D1
D2
D3
D4
D5

Fig. 5.1.4: ALU Bitslice Schematic

Bit 0 of the ALU bitslice circuitry is different from all other bits of the ALU because it is connected to nonbitslice circuitry which feeds it carry-in, '0', or S131 to the SHL/SHLC/ROL input associated with the *D3* control signal. This Bit 0 input circuitry is shown in Fig. 5.1.5(a). The SHL/SHLC/ROL input is attached to Bi-1' (S1i-1') for all other inputs. Bits 0-3 contain circuitry for Q0-3, the 4-bit *Quick* (Qi) or immediate operand used by AddQ and SubQ instructions and associated with the *D2* control signal. The *Qi* input is hardwired to ground for bits 4-31.

Bits 8-31 require an additional MUX for the byte insertion and extraction instructions (BIN, BEX) associated with the *D2* control signal. In place of this MUX, bits 0-7 hardwire the BIN/BEX input to the S1 bus.

Bit 31 is different than the other bits because a) shifts and rotates from the right come from nonbitslice circuitry, and b) In the event of a SET condition instruction, the condition is asserted onto bit 31 of the DST bus in place of the result of the ALU operation. The SHR/ROR input, associated with the *D3* control signal, is connected to the output of the bit 31 support circuitry detailed at the bottom of Fig. 5.1.5(b). The MUX associated with the *D6* control signal is unique to Bit 31 and provides an input for the *Cond* signal generated by the circuitry of Fig. 5.1.5(b).

5.1.4. Condition Tester

The condition tester tests for the arithmetic conditions specified by ARC's SET COND instruction. The condition tester also latches the overflow and carry conditions. When the SET instruction is executed, the output of the condition tester is passed to bit 31 of the DST bus so that the condition is stored in bit 31 of a register. The 8 conditions are selected by the 4-bit COND Field of the SET instruction as listed in Table 4.1.

Figure 5.1.7 is a floorplan of the condition tester as well as a floorplan of the bit 31 support circuitry. The floorplan shows the relative positions of functional blocks,

NON-BITSLICE ALU SUPPORT CIRCUITRY

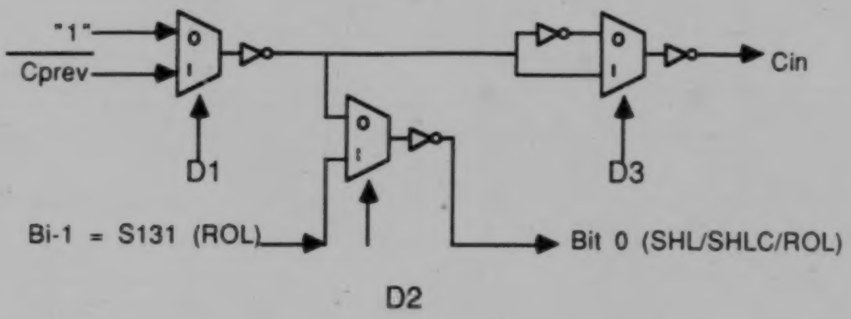

"1"
Cprev
0
1
D1
D2
D3
Cin
Bi-1 = S131 (ROL)
Bit 0 (SHL/SHLC/ROL)

Fig. 5.1.5 a): Bit 0 Input Circuitry

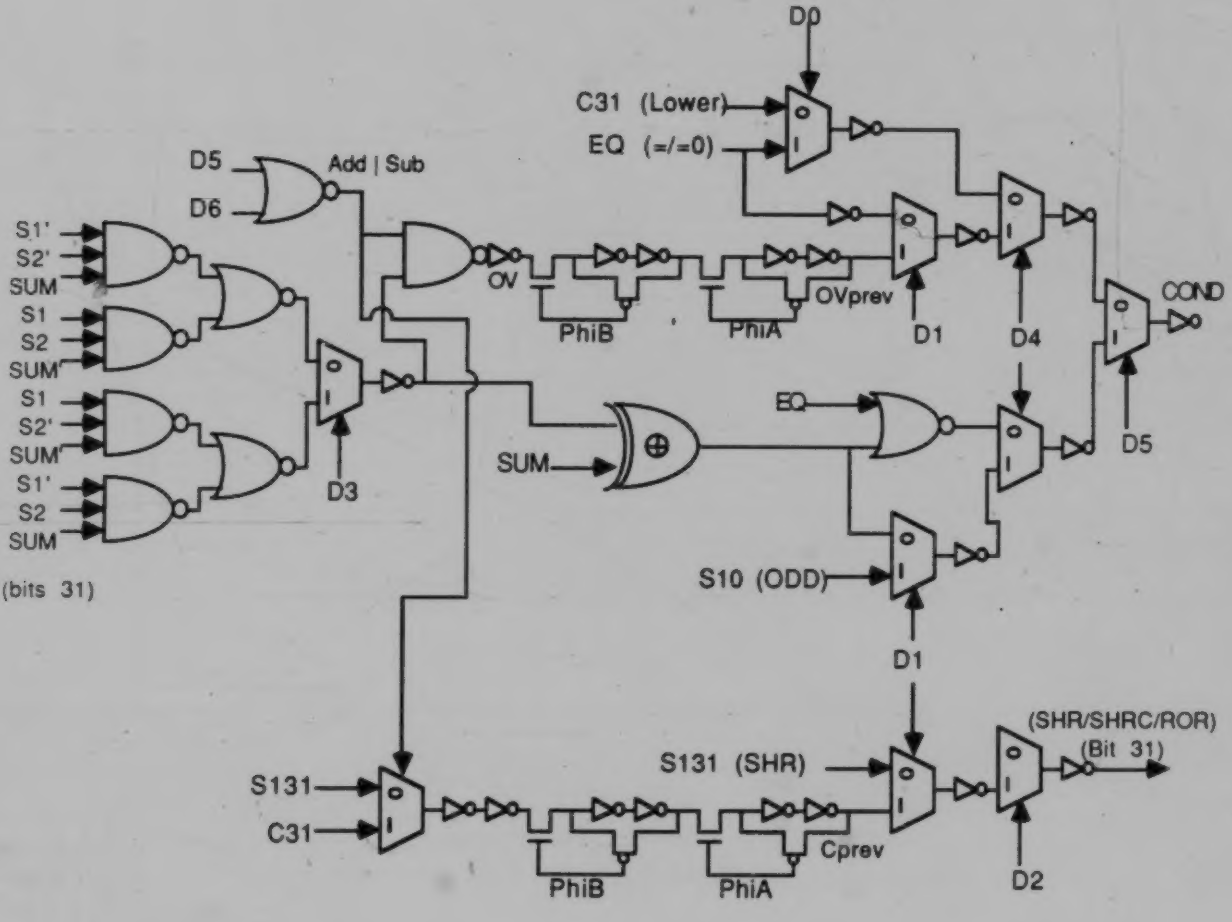

D0
C31 (Lower)
EQ (=/=0)
D5
D6
Add | Sub
S1'
S2'
SUM
S1
S2
SUM
S1
S2'
SUM
S1'
S2
SUM
(bits 31)
OV
PhiB
PhiA
OVprev
D1
D4
COND
D3
EQ
SUM
D5
S10 (ODD)
D1
(SHR/SHRC/ROR)
(Bit 31)
S131 (SHR)
S131
C31
Cprev
PhiB
PhiA
D2

Fig. 5.1.5 b): Condition Tester and Bit 31 Support Circuitry

ALU Transistor Schematics

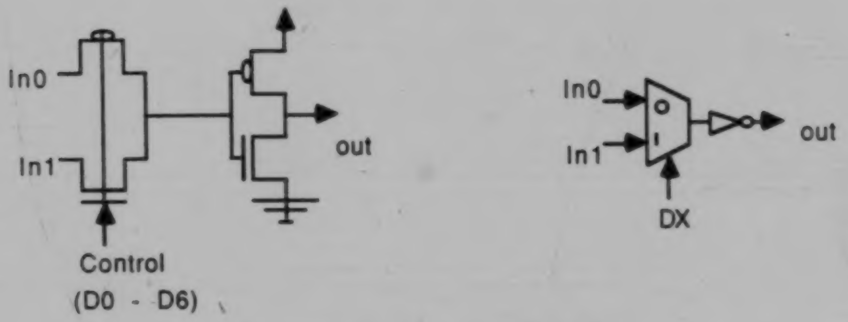

In0
In1
out
Control
(D0 - D6)
In0
In1
0
1
out
DX

Fig. 5.1.6 a): 2 to 1 Inverting MUX

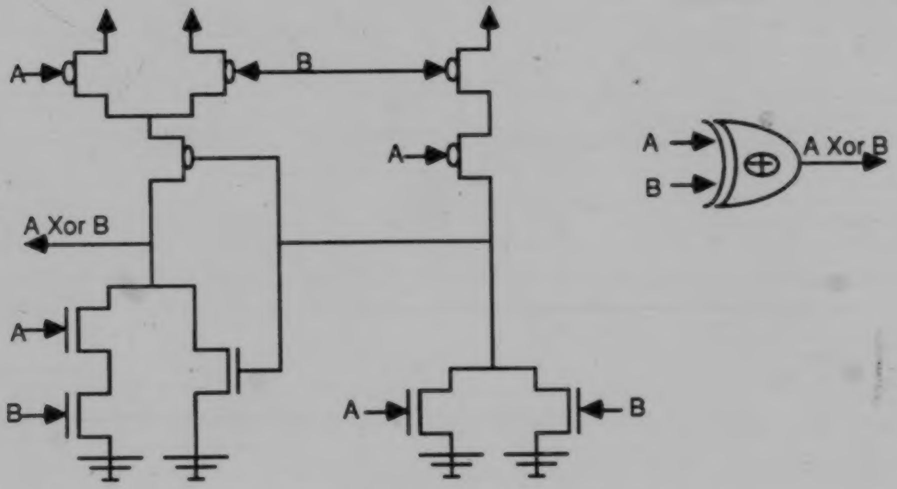

A
B
A
A Xor B
A
B
A
B
A
B
A Xor B

Fig. 5.1.6 b): 2 Input XOR

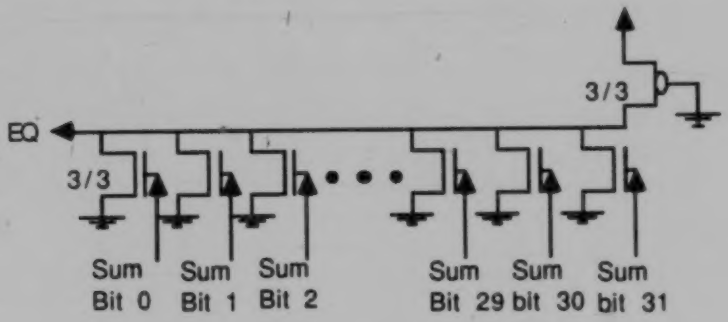

3/3
EQ
3/3
Sum Bit 0
Sum Bit 1
Sum Bit 2
Sum Bit 29
Sum bit 30
Sum bit 31

Fig. 5.1.6 c): EQ Line Circuitry

inputs, and outputs. The inputs to the condition tester from the ALU or adder include S10 (S1 bit0), EQ, C31 (Carry from bit 31), Sum31 (Sum bit 31), S131, and S231. The EQ line is a wired nor of the output stage from each of the ALU bitslices such that the EQ line is only high when all outputs are zero. The Condition tester also receives control signals (D0-D6) from the ALU decoder.

The schematic of the condition tester is portrayed in Fig. 5.1.5(b). The condition tester is implemented using 2 input inverting MUXes, NAND gates, NOR gates, D latches and an XOR gate. Transistor level schematics are given in Fig 5.1.6 for the inverting MUX, XOR gate, and EQ line circuitry. These implementations are more efficient than the traditional gate implementations of these functions.

5.1.5. Other Non-Bitslice ALU Circuitry

Other non-bitslice circuitry includes rotate, shift and carry inputs to bit 0 and bit 31 as well as the condition input to bit 31. This circuitry is implemented outside of the bitslice so that the pitch of bits 0 and 31 of the ALU bitslices could match that of the register file.

The bit 0 non-bitslice support circuitry aids in the implementation of shift- and rotate-left type instructions such as SHL (Shift Left), SHLC (Shift Left with Carry), and ROL (Rotate Left). A schematic for the bit 0 non-bitslice support circuitry is given in Fig. 5.1.5(a).

The bit 31 non-bitslice support circuitry aids in the implementation of shift- and rotate-right instructions such as SHR (Shift Right), SHRC (Shift Right with Carry), and ROR (Rotate Right). A floorplan for this circuitry is given in Fig 5.1.7 and a schematic is given in Fig 5.1.5(b). This floorplan is provided to document the physical design and facilitate layout inspection and modification.

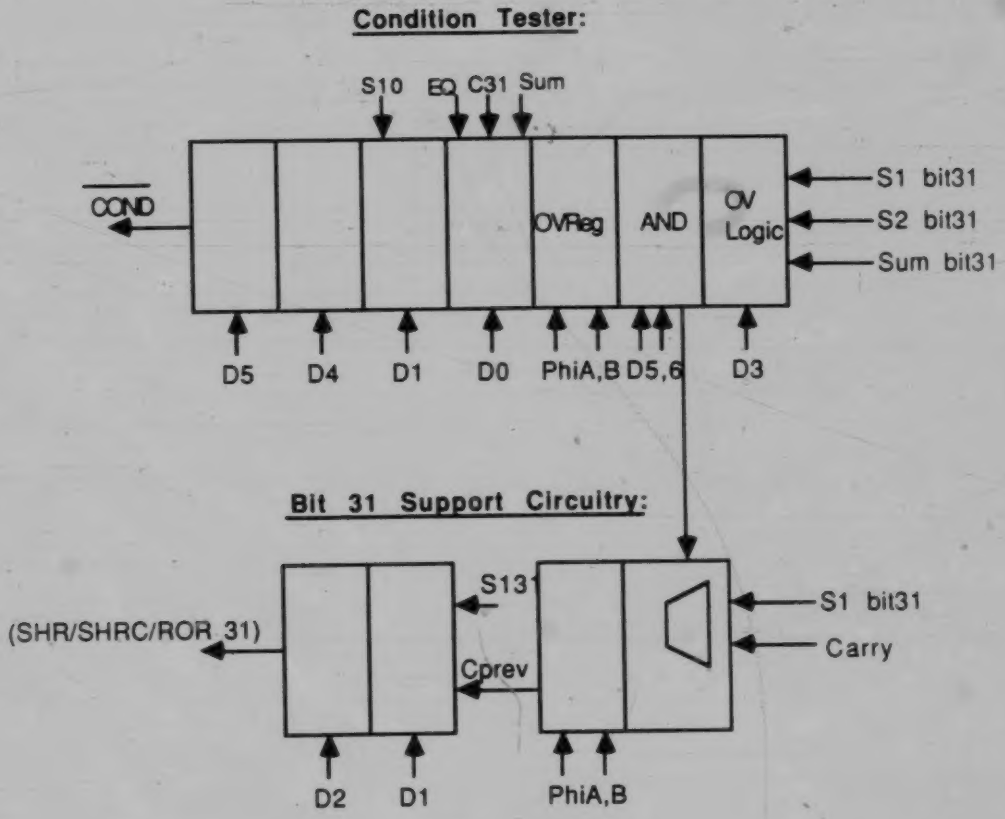

Condition Tester:
S10
EQ
C31
Sum
COND
OVReg
AND
OV Logic
S1 bit31
S2 bit31
Sum bit31
D5
D4
D1
D0
PhiA,B
D5,6
D3
Bit 31 Support Circuitry:
(SHR/SHRC/ROR 31)
S131
Cprev
S1 bit31
Carry
D2
D1
PhiA,B

Fig. 5.1.7: Floorplan of Condition Tester and Bit 31 Support Circuitry

5.2. Register File

The floorplan of the ARC Register File is depicted in Fig. 5.2.1. The Register File is composed of 3 sections: 1) a register array of 32 4-port 32-bit registers, 2) a set of 4 subwindow decoders, and 3) subwindow enable circuitry for each of the 4 subwindows. The Register File can also be considered to be composed of 4 subwindows (W0-W3). Each subwindow contains 8 32-bit registers, decoders for the 8 registers, and subwindow enable circuitry which determines if the subwindow is currently active.

Subwindows are paired to form overlapping register windows similar to the Berkeley RISC processors mentioned in Chapter 2. The subwindow select/enable circuitry is a set of PLAs which determine which subwindow decoders are to be activated, thereby facilitating the overlapping register feature of ARC. The first version of ARC has a window size of 16 registers with an overlap of 8.

The registers within the register array have 5 ports and service 4 different busses to interface to the ALU and IOCU bitslice circuitry. [2] The high number of ports and busses are necessary to support novel ARC features such as the background register saves and restores. The register array is connected to the ALU bitslice circuitry via 3 unidirectional busses, S1, ˉS2, and DST. The S1 and ˉS2 busses carry the contents of 2 registers to the ALU and the DST bus returns the result of an ALU operation to a register. The register array is connected to the IOCU bitslice circuitry via the unidirectional S1 bus and the bidirectional MMU bus. The S1 bus carries the contents of a register (usually representing a memory address) via the IOCU bitslice circuitry for dispatch out the OUT bus. The MMU bus is bidirectional and can be conditionally discharged by the IOCU bitslice circuitry or by a register. The direction of the MMU bus is determined by the M_O and M_I lines. If the M_O line is high, a register will

---

[2] The 5 register ports are mapped to 4 busses because the MMU bus is bidirectional and therefore services 2 register ports.

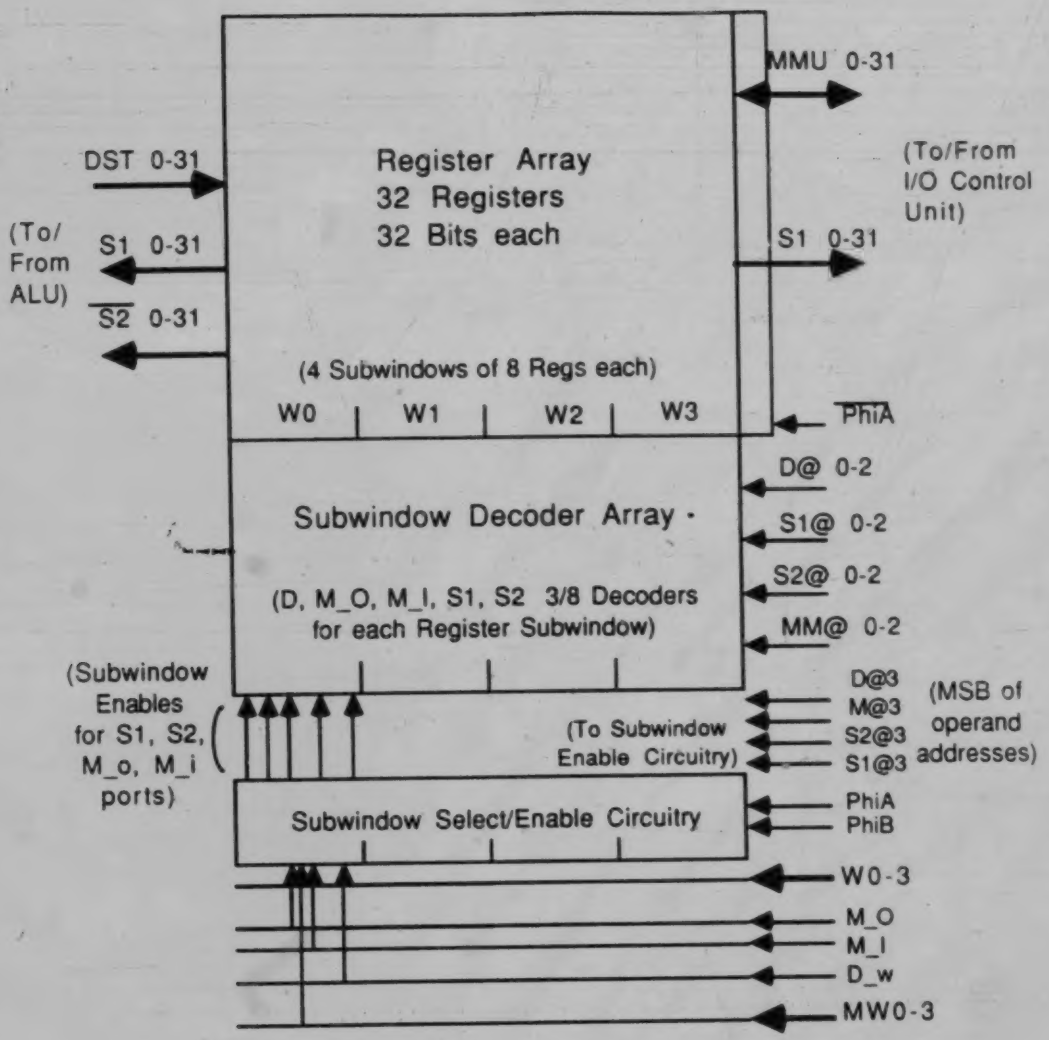

MMU 0-31
(To/From
I/O Control
Unit)
S1 0-31
DST 0-31
(To/
From
ALU)
S1 0-31
S2 0-31
Register Array
32 Registers
32 Bits each
(4 Subwindows of 8 Regs each)
W0
W1
W2
W3
PhiA
D@ 0-2
S1@ 0-2
S2@ 0-2
MM@ 0-2
Subwindow Decoder Array
(D, M_O, M_I, S1, S2 3/8 Decoders
for each Register Subwindow)
(Subwindow
Enables
for S1, S2,
M_o, M_i
ports)
D@3
M@3
S2@3
S1@3
(MSB of
operand
addresses)
(To Subwindow
Enable Circuitry)
PhiA
PhiB
Subwindow Select/Enable Circuitry
W0-3
M_O
M_I
D_w
MW0-3

Fig. 5.2.1: Register File Floorplan

conditionally discharge the MMU bus. If the M_I line is high, the value on the IN bus will be written onto the MMU bus.

The subwindow decoders are essentially a set of 3-to-8 decoders for each of the 5 register ports (D, M_O, M_I, S1, S2). The decoder inputs are 3-bit addresses for each of the register busses (D@0-2, MM@0-2, S1@0-2, S2@0-2). These 3-bit addresses are shown to the right of the subwindow decoders in Fig. 5.2.1.

The floorplan of the subwindow enable/select circuitry is shown at the bottom of Fig. 5.2.1. The inputs to the subwindow enables include Window selects (W0-3, MW0-3), the most significant bit of register operand addresses (S1@3, S2@3, D@3, M@3), clocks (PhiA, PhiB), and register write enables (D_w, M_I, M_O). The current ALU window (selected by W0-3) is controlled by the 4-bit Current Window Pointer (CWP) of the IOCU. The MMU window (selected by MW0-3) is also controlled by the CWP while executing Transport or Control Instructions. During "background" register saves and restores, MW0-3 is controlled by the 4-bit Save Window Pointer (SWP). The CWP is implemented as a circular shift register which is shifted right after each JSR instruction and shifted left after each RET instruction. The SWP references the register window next (or currently) to be saved. The MMU window (MW0-3) is not necessarily the same as the current window (W0-3) because the IOCU is charged with the task of saving and restoring unused RF activations.

The design of the Register File for the ARC was first tackled during the fall of 1986 as a course project. Since then, the register cells were redesigned by Emil Girczyc to conserve space. The Decoder and enable circuitry was also redesigned to conserve space during the fall of 1987.

5.2.1. Register Array

The register array is made up of 2 types of cells, 1) the register-pair cell, and 2) the register precharge cell. The register pair cell is essentially a bitslice for 2 registers.

Registers were paired for two significant reasons: 1) to allow the ALU to have a larger pitch, and 2) to minimize the capacitance of the register file busses (Bus lines are shorter). The register precharge cell contains the precharge transistors for the S1, ~S2, and MMU busses in addition to the interface between the register array and the I/O Control bitslice circuitry.

The register-pair cell floorplan of Fig. 5.2.2 indicates the relative positions of the busses running horizontally and the decoder signals running vertically through the cell. The register-bit schematic of Fig. 5.2.3 shows the various ports, control signals, bus connections, transistor connections and sizes of a register bit. Each register contains 5 ports (S1, S2, M_O, M_I, DST) controlled by the decoder signals from the bottom of the register-pair cell. When a decoder signal is high, the corresponding port facilitates a transfer from register to bus or vice-versa. When an *S1* decoder signal is high, the S1 port of that particular 32-bit register conditionally discharges the S1 bus. Only one *S1* decoder signal may be high at one time for all of the registers in the array. Similarly with the *S2* decoder signal. The *M_O* and *M_I* decoder signals cause a MMU bus conditional discharge or register read respectively. Only one register may access the MMU bus at any time. When a *DST* decoder signal is high, the contents of the 32-bit DST bus are transferred into the particular register referenced by the high *DST* decoder signal. All dynamic busses (S1, ~S2, and MMU) are precharged during PhiA and conditionally discharged during PhiB (execution phase). Therefore the *S1*, *S2*, *M_I*, and *M_O* decoder signals can only be active during PhiB. In contrast, the DST bus is static. The output of the ALU is written onto the DST bus during PhiB and during the following PhiA. The contents of the DST bus may be loaded into a register during the PhiA. This extends the amount of time for the critical path by the time required to drive the DST bus or the length of PhiA (whichever is shorter).

The design of registers is important in any RISC because the performance of RISC processors is closely related to the number of registers. The register-pair is the

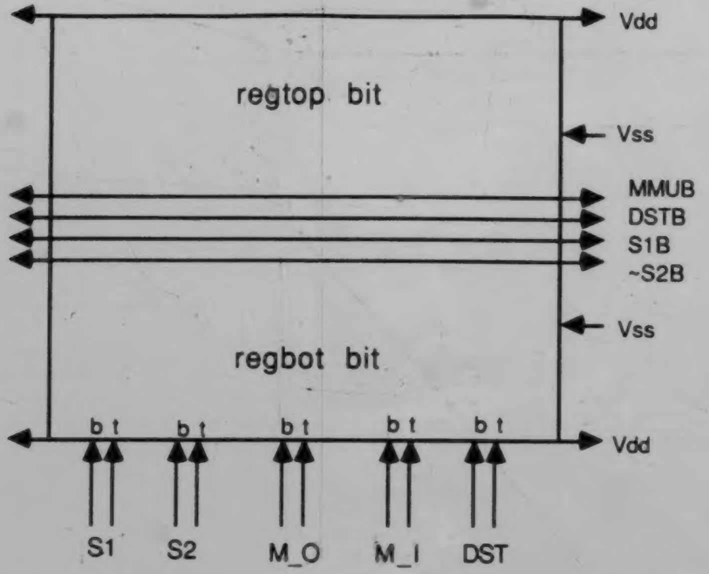
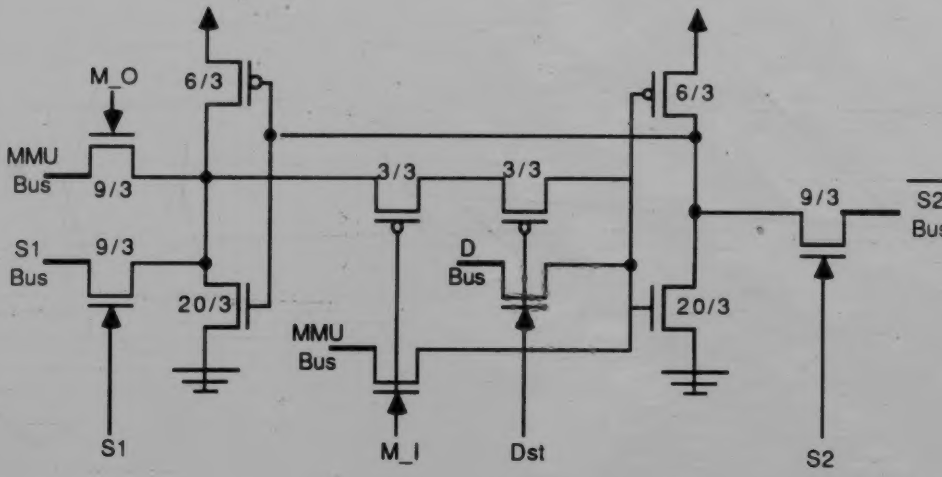

Vdd
regtop bit
Vss
MMUB
DSTB
S1B
~S2B
Vss
regbot bit
b t
b t
b t
b t
b t
Vdd
S1
S2
M_O
M_I
DST

Fig. 5.2.2: Register-Pair Cell Floorplan


M_O
MMU Bus
9/3
S1 Bus
9/3
6/3
20/3
3/3
3/3
D Bus
MMU Bus
6/3
20/3
9/3
$\overline{\text{S2}}$ Bus
S1
M_I
Dst
S2

Signal	Phase Asserted
S1	PhiB
S2	PhiB
M_O	PhiB
M_I	PhiB
Dst	PhiA

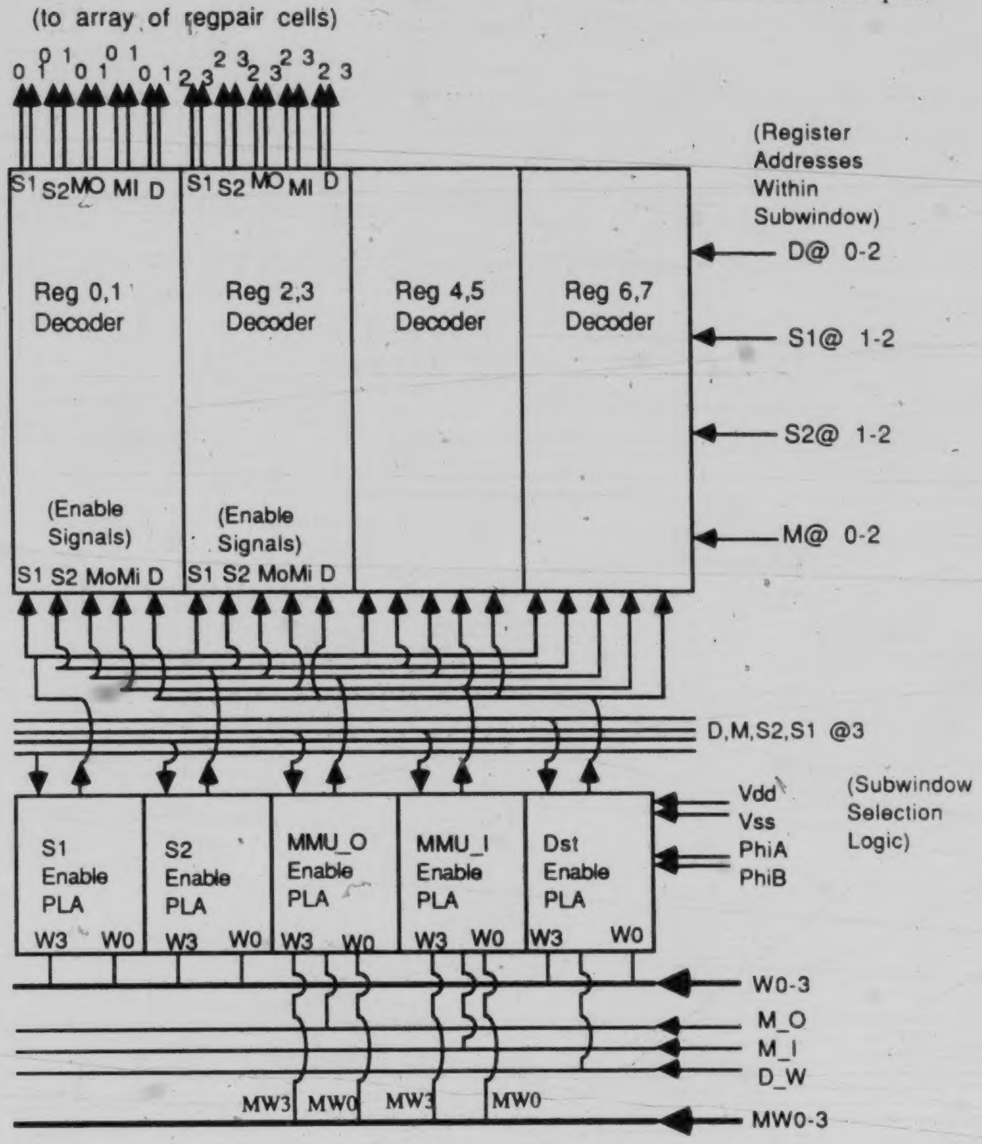
Fig. 5.2.3: Register Bit Schematic

densest and most technology dependent cell in ARC. In the design of the register file, tradeoffs had to be made between a) register bit size, b) number of registers, c) register speed, and d) capacitive load of registers on the busses. The register bit size is critical because this dictates the number of registers that will fit on the chip. The register speed can be altered by increasing the width of the bus discharge transistors but this also increases the register bit size and capacitive load on the busses. The higher the capacitive load on the busses, the more time is required to precharge and discharge each of the busses. Capacitive load is increased as the number of registers or the drive (size) of register ports increases. The schematic of a register bit given in Fig. 5.2.3 gives suggested aspect ratios for the Northern Telecom CMOS3 process but optimization must be redone for reimplementation in a more aggressive technology.

5.2.2. Subwindow Decoder

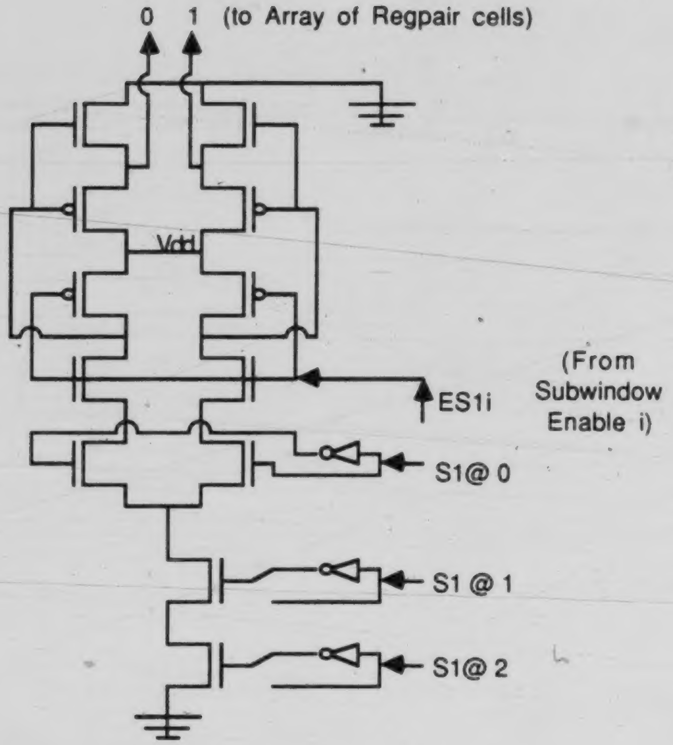
The subwindow decoder floorplan of Fig. 5.2.4 shows the decoder as well as subwindow enable circuitry for an 8-register subwindow. A subwindow decoder is shown at the top of Fig. 5.2.4 and is divided into 4 decoders - a decoder for each of the register pairs in the subwindow. In the floorplan, the register port (decoder) signals are outputs at the top, register addresses are inputs from the right, and enable signals are inputs from the bottom. The register decoder signals control the register ports and correspond to the *S1, S2, M_O, M_I,* and *DST* decoder signals described previously in Section 5.2.1. The register addresses are extracted from instructions and are labelled *S1@, S2@, M@, and D@. Only the 3 least significant bits (bits 0-2) of the register addresses are inputs to the subwindow decoders. These 3-bit addresses will select 1 of the 8 registers each within the register subwindow providing the subwindow is enabled by the enable signals from the enable PLAs.*

The partial schematic of a decoder cell is given in Fig. 5.2.5. This schematic shows the relative placement of transistors and signals for an example decoder pair.


(to array of regpair cells)
0 1 0 1 0 1 0 1 0 1
2 3 2 3 2 3 2 3 2 3
S1 S2 MO MI D
S1 S2 MO MI D
Reg 0,1 Decoder
Reg 2,3 Decoder
Reg 4,5 Decoder
Reg 6,7 Decoder
(Enable Signals)
(Enable Signals)
S1 S2 MoMi D
S1 S2 MoMi D
(Register Addresses Within Subwindow)
D@ 0-2
S1@ 1-2
S2@ 1-2
M@ 0-2
D,M,S2,S1 @3
S1 Enable PLA
S2 Enable PLA
MMU_O Enable PLA
MMU_I Enable PLA
Dst Enable PLA
W3 W0
Vdd
Vss
PhiA
PhiB
(Subwindow Selection Logic)
W0-3
M_O
M_I
D_W
MW3
MW0
MW3
MW0
MW0-3

(Above Example Depicts Subwindow 3 Decoders/Enables)

Fig. 5.2.4: Subwindow Decoder Floorplan


0 1 (to Array of Regpair cells)
Vdd
ES1i
(From Subwindow Enable i)
S1@ 0
S1 @ 1
S1@ 2

(Above Example depicts typical decoder for S1 Register Port)
(Example decodes addresses 0 and 1)

Fig. 5.2.5: Example Decoder Schematic

Enable PLA Equations:

ES1 = (Wi & ~S1@3 & ØB) | (Wi-1 & S1@3 & ØB)

ES2 = (Wi & ~S2@3 & ØB) | (Wi-1 & S2@3 & ØB)

EMOi = (MWi & ~M@3 & ØB & M_O) | (MWi-1 & M@3 & ØB & M_O)

EMIi = (MWi & ~M@3 & ØB & M_I) | (MWi-1 & M@3 & ØB & M_I)

EDi = (Wi & ~D@3 & ØA & D_W) | (Wi-1 & D@3 & ØA & D_W)

Fig. 5.2.6: PLA Equations for Enable Circuitry

The decoder cell is essentially a collection of NAND-form decoder-pairs for each of the 5 pairs of register decoder signals: *S1*, *S2*, *M_O*, *M_I*, and *D*. The pairing of decoders results in some space savings because the addresses of the pair differ only in the least significant bit of the address (Note that S1@1 and S1@2 are decoded by 2 transistors instead of 4). As well, the decoder signals are paired to match those of the register array. The enable signals, *ES1i*, *ES2i*, and *EDi* (i indicates subwindow), hold the register decoder signals low when the subwindow is not active. The enable signals are described in the following section.

5.2.3. Subwindow Enables

The subwindow enable circuits facilitate the overlapping register feature of ARC by managing the selection of a subwindow. As well, these circuits ensure that the register decoder signals are enabled during the correct clock phases.

The subwindow enable floorplan is shown at the bottom of Fig. 5.2.4. The inputs to the subwindow decoders include Window selects (W0-3, MW0-3), the most significant bit of register operand addresses (S1@3, S2@3, D@3, M@3), clocks (PhiA, PhiB), and register write enables (D_w, M_I, M_O).

The 4-bit Window select signal, *W0-3*, indicates the subwindow pair which is the current ALU window (for S1, S2, and DST addresses). The subwindow within the subwindow pair is selected by S1@3, S2@3, and D@3.

The 4-bit MMU Window select signal, *MW0-3*, selects the subwindow pair which is the I/O (or MMU) window. The 4-bit *MW0-3* signal is the same as *W0-3* during transport or control instructions but during background register saves/restores, *MW0-3* will point to another window.

The most significant bit of each of the register addresses extracted from an instruction field is used by the subwindow enable PLA to select one of the two subwindows of the current 16-bit overlapping register window.

The clock signals, *PhiA* and *PhiB*, are needed to ensure that the register decoder signals are enabled during the correct phases. *S1*, *S2*, *M_O*, and *M_I* register decoder signals are enabled during PhiB (execution phase) while the DST register decoder signal is enabled during PhiA.

The register write enable signals, *D_w*, *M_I*, and *M_O*, are input to the subwindow enables by the IOCU. When the *D_w* signal is low, writes from the DST bus to the register file are disabled and changes to the overflow and carry flags are suppressed. This is used to conditionally cancel the current ALU instruction. (The *D_w* signal is conditionally pulled low by the preceding SKPONC operation.) The *M_I* and *M_O* signals select the direction of the bidirectional MMU bus.

The PLA equations for the various subwindow enable circuits are given in Fig. 5.2.6.

5.3. The I/O and Control Unit (IOCU)

The IOCU is responsible for I/O functions such as the interface to ARC's unidirectional IN and OUT busses and is also responsible for Control functions such as instruction extraction and decoding, register window control, and *background* register saves and restores. The IOCU also supports unidirectional busses, two instructions per word, pipelined memory access, and the Memory Control Unit with an external PC - all novel features of ARC presented in Chapter 3. In addition, transport instructions such as LoaD, STore, LDPC, and LDI and control instructions such as branches and the conditional skip instruction (SKPONC) are the responsibility of the IOCU.

A block diagram of the IOCU is given in Fig. 5.3.1. This block diagram shows the relative positions and approximate sizes of functional blocks. The IOCU can be considered as being composed of four functional blocks:

1 I/O Bitslice Circuitry

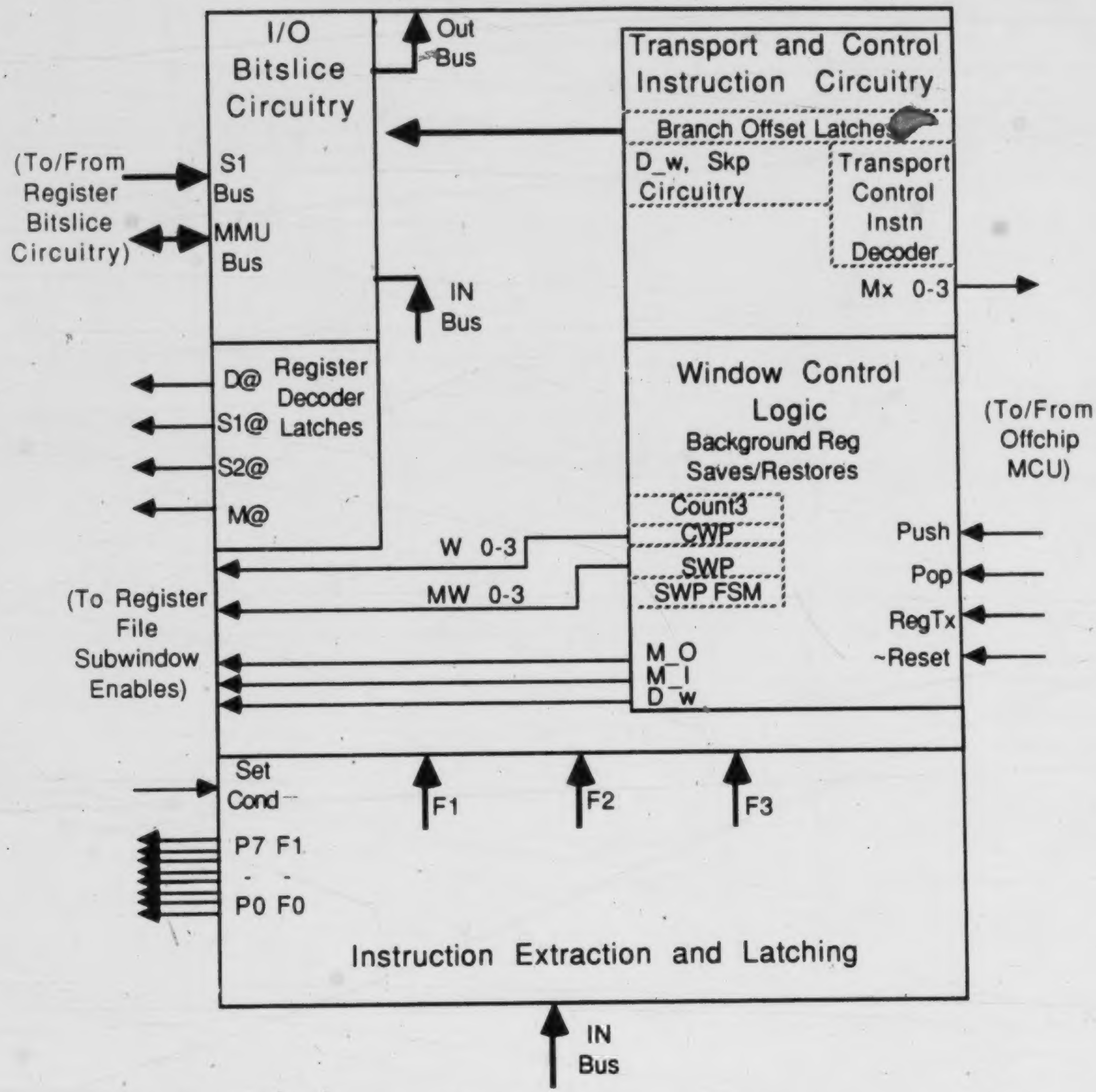

I/O Bitslice Circuitry
Out Bus
Transport and Control Instruction Circuitry
Branch Offset Latches
D_w, Skp Circuitry
Transport Control Instn Decoder
(To/From Register Bitslice Circuitry)
S1 Bus
MMU Bus
IN Bus
Mx 0-3
Register Decoder Latches
D@
S1@
S2@
M@
Window Control Logic
Background Reg Saves/Restores
(To/From Offchip MCU)
Count3
CWP
SWP
SWP FSM
W 0-3
MW 0-3
Push
Pop
RegTx
~Reset
(To Register File Subwindow Enables)
M_O
M_I
D_w
Set Cond
F1
F2
F3
P7 F1
P0 F0
Instruction Extraction and Latching
IN Bus

Fig. 5.3.1: I/O and Control Unit Block Diagram

2 Instruction Extraction and Latching Circuitry
3 Transport and Control Instruction Circuitry
4 Window Control Logic.

Inputs to the IOCU include the 32-bit IN bus from off-chip, the 32-bit S1 and MMU busses from the register file, the *Push*, *Pop*, *RegTx*, and *Resetbar* signals from off-chip, and the *Set Cond* signal from the ALU decoder. Data outputs of the IOCU include the 32-bit OUT bus to off-chip and the 32-bit MMU bus to the register file. Signal outputs of the IOCU include register and window addresses to the register file (D@, M@, S1@, S2@, W0-3, MW0-3), register file control signals (M_O, M_I, D_w), the instruction opcode to the ALU (P0-P7), and a 4-bit signal to the External Memory Control Unit (Mx0-3).

5.3.1. I/O Bitslice Circuitry

The Floorplan of the I/O bitslice circuitry is as shown at the top left of Fig. 5.3.1. This circuitry is located adjacent to and directly connected to the bitslice circuitry of the Register File. This circuitry provides a switchbox for the S1 and MMU busses of the register file, the IN bus, the OUT bus, and the branch offset (one of the fields of relative branch instruction). The schematic for the I/O bitslice circuitry is given at the top of Fig. 5.3.4.

Inputs to the I/O bitslice circuitry include the following control signals from the I/O Control Decoder PLA:

S1Out:

causes the value on the S1 bus to be written to the OUT bus. This signal is asserted when the contents of a register (interpreted as a memory address) is to be written to the OUT bus. This signal is asserted during the address phase of LoaD, STore, JUMP, or JUMP-to-SubRoutine (JSR) instructions.

mmOut:

causes the value on the MMU bus to be written to the OUT bus. This signal is asserted when the contents of a register (interpreted as data) is to be written to the OUT bus. This signal is asserted during the data phase of STore instructions and during background register saves.

mmIn:

causes the value on the IN bus to be read into the MMU bus. This signal is asserted during the data phase of all LoaD instructions and during register restores. This signal allows data to be loaded from the IN bus via the MMU bus into a register. The *mmIn* signal could also be delayed to increase the pipe length of data fetches from memory to allow pipelined memory access as described in section 3.4. The implementation would involve inserting variable delays controlled by signals from the MCU. This feature was not implemented in the first version of ARC.

brout:

causes the 8-bit *branch offset* field of a relative branch instruction to be written to the OUT bus.

Some restrictions apply with regard to when the above signals may be asserted. All data transfers via the IN and OUT bus occur during PhiB. Obviously only one value may be written to the OUT bus at a time. As well, data may be transferred in through the IN bus during odd clock phases (Phi1B, Phi3B) because instruction pairs are transferred over the IN bus during even clock phases. Conflicts are resolved according to the pipeline and resource scheduling diagrams of Fig. 4.11.

5.3.2. Instruction Extraction and Latching Circuitry

Fig. 5.3.2 shows instruction extraction and latching circuitry. This circuitry is composed of several functional blocks including the INMUX (INstruction MUX), latches for the instruction, as well as multiplexers and latches to extract operands from the instruction.

The instruction multiplexer separates instruction pairs into two 16-bit instructions. During each even phase (Phi0B, Phi2B), the 32-bit word on the IN bus is expected to contain a pair of instructions. The first halfword instruction (IN0-15) is latched into the instruction latch and thereby enters the instruction pipeline. The second halfword instruction (IN16-31) is latched within the INMUX until the next PhiB phase at which time it is latched into the instruction latch and then enters the instruction pipeline one cycle behind the first halfword instruction.

The instruction within the instruction latch is divided into 4 fields labelled F0, F1, F2, and F3 in Fig. 5.3.2. These fields correspond to different operand types depending on the instruction type as specified by the instruction formats given in Fig. 4.9. The instruction field extraction circuitry of the IOCU is mainly concerned with extracting the S1@, S2@, MMU@, and DST@ operands. The opcode information is the F0 and F1 fields, which are passed directly to the ALU and Transport/Control Instruction Decoders. To organize the instruction fields into the different operand types, multiplexers are used as shown in the middle of Fig. 5.3.2. Control signals from the ALU and I/O Control decoders (Set Cond, I/O Op) are used to select the extraction of operands from the instruction fields.

Once the S1@, S2@, MMU@, and DST@ operands of the instruction have been sorted, they are latched adjacent to the Register File decoder circuitry. The S1@, S2@, and MMU@ operands are queued in these latches until the next PhiB phase (as specified by the instruction pipeline) at which time they drive the register decoder and enable circuitry. The DST@ operand must be queued until the following PhiA.

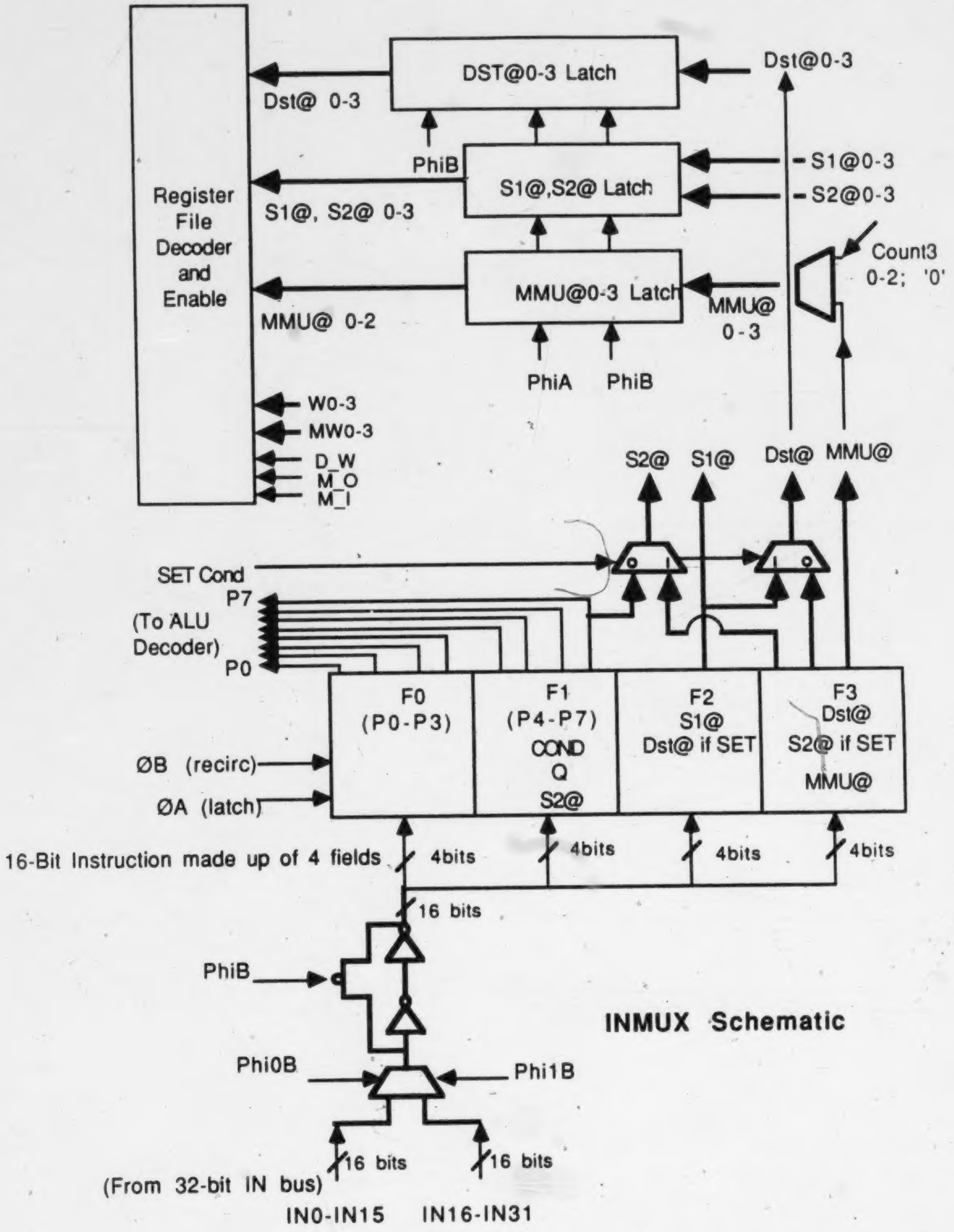

DST@0-3 Latch
Dst@0-3
Dst@ 0-3
PhiB
S1@,S2@ Latch
- S1@0-3
- S2@0-3
Register File Decoder and Enable
S1@, S2@ 0-3
Count3 0-2; '0'
MMU@0-3 Latch
MMU@ 0 - 3
MMU@ 0-2
PhiA
PhiB
W0-3
MW0-3
D_W
M_O
M_I
S2@
S1@
Dst@
MMU@
SET Cond
P7
(To ALU Decoder)
P0
F0 (P0-P3)
F1 (P4-P7) COND Q S2@
F2 S1@ Dst@ if SET
F3 Dst@ S2@ if SET MMU@
ØB (recirc)
ØA (latch)
16-Bit Instruction made up of 4 fields
4bits
4bits
4bits
4bits
16 bits
PhiB
INMUX Schematic
Phi0B
Phi1B
16 bits
16 bits
(From 32-bit IN bus)
IN0-IN15
IN16-IN31

Fig. 5.3.2: Instruction Field Extraction/Multiplexing and Latching

5.3.3. Transport and Control Instruction Circuitry

The IOCU is responsible for the I/O and control instructions given in the table at the top of Fig. 5.3.3. The Transport and Control Instruction circuitry is composed of the I/O Control Decoder PLA, D_w circuitry, and other logic which performs preliminary instruction decoding.

The I/O Control Decoder PLA shown in Fig. 5.3.4 is responsible for decoding the I/O and control instructions into various control signals. The I/O decoder inputs and output control signal equations are listed in Fig. 5.3.3. A short explanation of each of these signals follows:

S1Out:

This signal is asserted if a LoaD, STore, JUMP, or JSR instruction is decoded. This signal is a decoder control signal passed to the I/O bitslice circuitry.

mmOut:

This signal is a decoder control signal used in the I/O bitslice circuitry. *mmout* is asserted if a STore instruction is decoded.

mmIn:

This signal is a decoder control signal sent to the I/O bitslice circuitry. *mmIn* is asserted if a LoaD (LD), LoaD Immediate (LDI), or LoaD Special Register (LDSR) instruction is decoded. This signal could be delayed to increase the pipe length of data fetches from memory, allowing the processor to elegantly react to a cache miss.

brOut:

This signal is a decoder control signal passed to the I/O bitslice circuitry. The *brOut* signal is asserted if a relative BRANCH instruction is decoded.

Mx0-3:

Instruction	P0-P3	P4-P7	MX0-MX3
LD	1111	0001	0001
LDSR	1111	10XX	01XX
ST	1111	0011	0011
LDI	1111	1101	1101
JMP	1111	1011	1011
JSR	1111	1001	1001
BRA	1111	1010	1010
RET	1111	1000	1000
SKPONC	1111	0000	0000
non-mmu	XXXX	XXXX	1111
Pushmode	XXXX	XXXX	1100
Popmode	XXXX	XXXX	1110

I/O Control Decoder PLA Inputs

MMUOP = P0*P1*P2*P3

MMUINST = MMUOP * $\overline{\text{SKP}}$

SKP = SKPONC(pravious) * S131

PhiB ↑ ↑ PhiA

I/O Control Decoder PLA Output Equations

S1OUT = LD | ST | JMP | JSR

S1OUT = 0001 | 0011 | 1011 | 1001

S1OUT = X0X1 = P5' * P7 * MMUINST

MMOUT = ST = 0011 = P4'*P5'*P6*P7*MMUINST

MMIN = LD | LDSR | LDI
MMIN = 0001 | 01XX | 1101
MMIN = (P4'*P5'*P6'*P7 | P4'*P5 | P4*P5*P6'*P7) * MMUINST

BROUT = 1010 = P4*P5'*P6*P7' * MMUINST

SKPONC = 0000 = P4'*P5'*P6'*P7' * MMUINST

LDI = 1101 = P4*P5*P6'*P7 * MMUINST

JSR = 1001 = P4*P5'*P6'*P7 * MMUINST

RET = 1000 = P4*P5'*P6'*P7' * MMUINST

MX0 = P4 | MMUINST'

MX1 = P5 | MMUINST'

MX2 = P6

MX3 = P7

Fig. 5.3.3: I/O Control Opcodes and Decoder PLA Equations

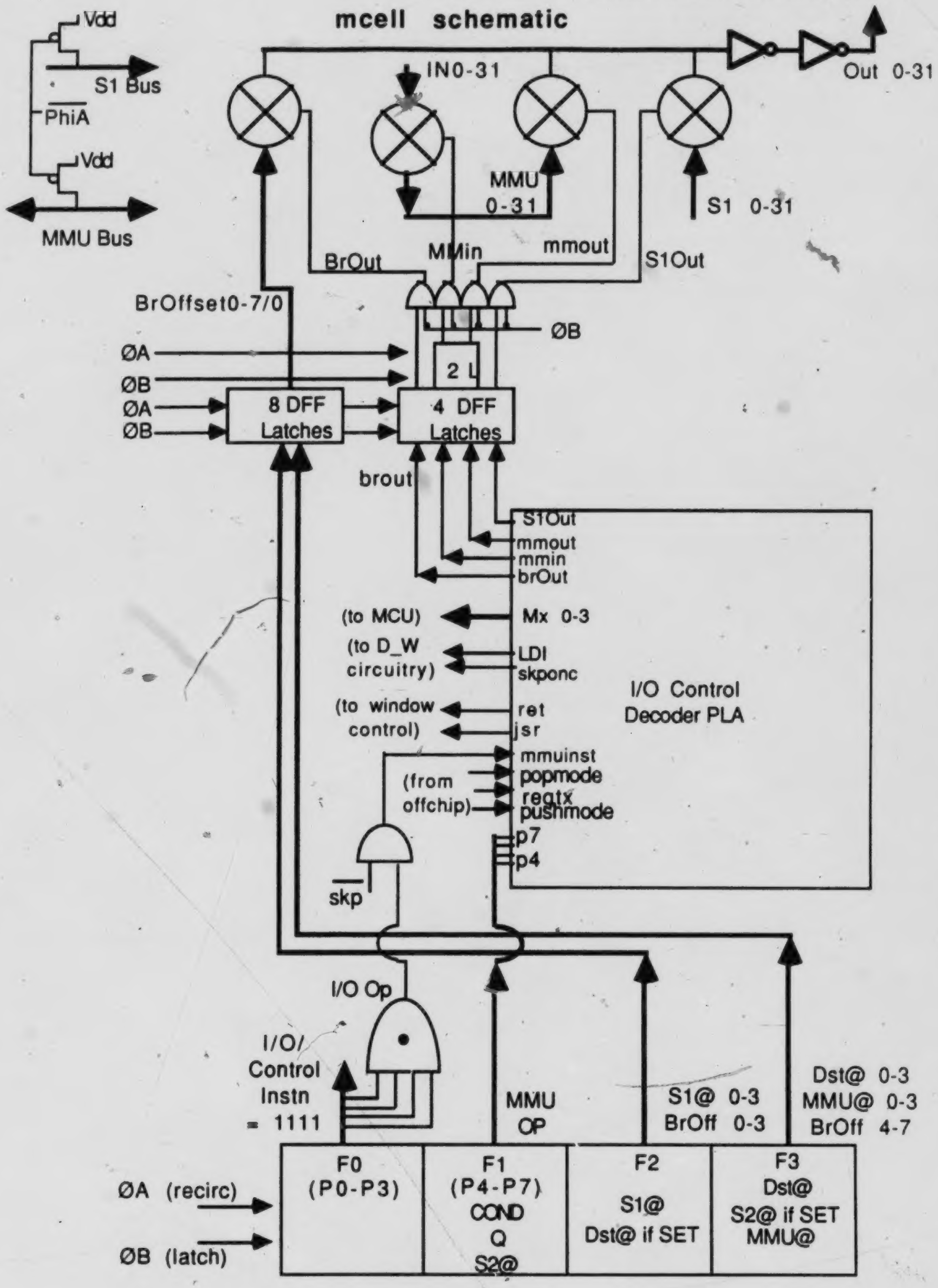

mcell schematic
Vdd
S1 Bus
PhiA
Vdd
MMU Bus
IN0-31
Out 0-31
MMU 0-31
S1 0-31
BrOut
MMin
mmout
S1Out
BrOffset0-7/0
ØB
ØA
ØB
ØA
ØB
2 L
8 DFF Latches
4 DFF Latches
brout
S1Out
mmout
mmin
brOut
(to MCU)
Mx 0-3
(to D_W circuitry)
LDI
skponc
(to window control)
ret
jsr
I/O Control Decoder PLA
mmuinst
popmode
regtx
pushmode
(from offchip)
p7
p4
skp
I/O Op
I/O/ Control Instn = 1111
MMU OP
S1@ 0-3
BrOff 0-3
Dst@ 0-3
MMU@ 0-3
BrOff 4-7
ØA (recirc)
ØB (latch)
F0 (P0-P3)
F1 (P4-P7) COND Q S2@
F2 S1@ Dst@ if SET
F3 Dst@ S2@ if SET MMU@

Fig. 5.3.4: I/O Control Operations Decoding and Latching

This 4-bit code is sent to the off-chip Memory Control Unit to let it know what I/O operations to expect from ARC.

LDI:

This signal is generated if a LoaD Immediate (LDI) instruction is detected. This signal is routed to the D_w circuitry to suppress execution of the next incoming instruction pair allowing the next word in the instruction stream to be treated as data rather than instruction.

SKPONC:

This signal is generated if a SKIP ON Condition (SKPONC) instruction is detected. The *SKPONC* signal is routed to the D_w circuitry to suppress execution of the next instruction. This instruction is used in conjunction with the BRANCH or JUMP instructions to emulate conditional branching.

RET:

The *RET* signal indicates that a RETurn (from subroutine) instruction is detected.

JSR:

The *JSR* signal indicates that a Jump (to SubRoutine) instruction is detected. The *JSR* and *RET* are passed to the Window Control circuitry.

The D_w circuitry is responsible for the *D_w* signal which enables writes from the DST bus to the Register File. Because the results of all ALU instructions are transferred via the DST bus to a register, the D_w circuitry is responsible for enabling (or disabling) ALU instructions. A schematic for the D_w circuitry is given in Fig. 5.3.5. The *D_w* signal is intended to be low when immediate data loads occur or when it becomes necessary to skip an instruction.

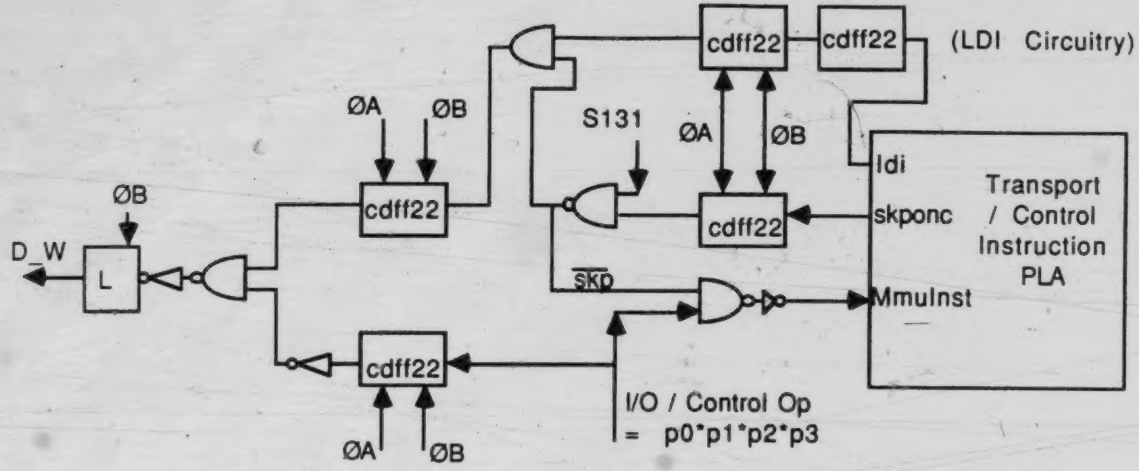

cdff22
cdff22
(LDI Circuitry)
ØA
ØB
S131
ØA
ØB
ldi
Transport / Control Instruction PLA
skponc
ØB
cdff22
cdff22
D_W
L
$\overline{\text{skp}}$
Mmulnst
cdff22
I/O / Control Op = p0*p1*p2*p3
ØA
ØB

Fig. 5.3.5: Control Operations D_w Circuitry

5.3.4. Window Control Logic.

The state of the overlapping register windows of ARC is controlled by a Current Window Pointer (CWP) and a Save Window Pointer (SWP). The concept of using a CWP and a SWP to implement overlapping register windows was pioneered by the Berkeley RISC group (see Section 2.7) The function of the window control logic of ARC is to manage the register file windows for ALU and I/O operations as well as for *background* register save/restore operations. The window control logic is composed of

a) two 4-bit circular shift registers which implement the CWP and the SWP.

b) a 3-bit counter for incrementing through the registers within a subwindow which is being saved/restored;

c) multiplexers for the MMU window and MMU address;

d) a finite state machine to help manage the *background* register saves and restores.

The floorplan of the window control logic is shown on the mid-right of Fig. 5.3.1.

The schematic for the 4-bit shift register (SR4), which makes up the heart of the CWP and SWP, is given in Fig. 5.3.6. The schematic for the CWP is given in Fig. 5.3.7. The CWP operates as follows:

> When the CWP is given a *Reset* signal, bit 0 (W0) of the SR4 is set to logic "1". The other bits of SR4, all latches and D flip-flops are set to logic "0".
>
> When a JSR instruction is detected by the decoder, the *JSR* signal is asserted to the CWP circuitry and latched into the D flip-flop. After an appropriate number of pipestage delays (to accomodate the delayed branch and pipeline features of ARC) the SR4 shift register receives the signal to shift up. Then the shift register shifts the logic "1" from W0 to W1 and a new register window becomes the *active* (current) window.

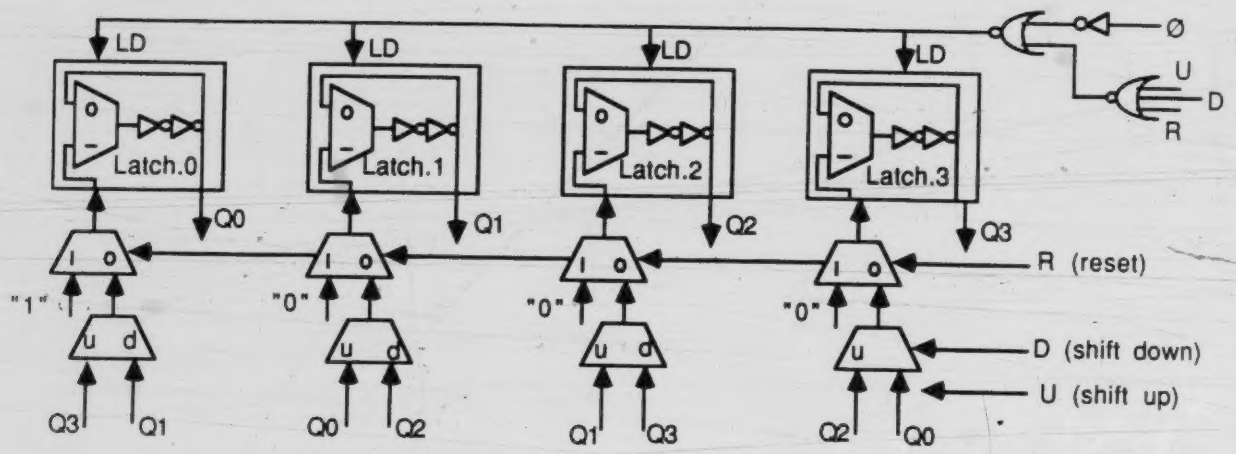

LD
LD
LD
LD
o
Latch.0
Latch.1
Latch.2
Latch.3
Q0
Q1
Q2
Q3
Ø
U
D
R
I
o
R (reset)
"1"
"0"
"0"
"0"
u
d
D (shift down)
U (shift up)
Q3
Q1
Q0
Q2
Q1
Q3
Q2
Q0

Fig. 5.3.6: 4-Bit Shift Register (SR4) Schematic

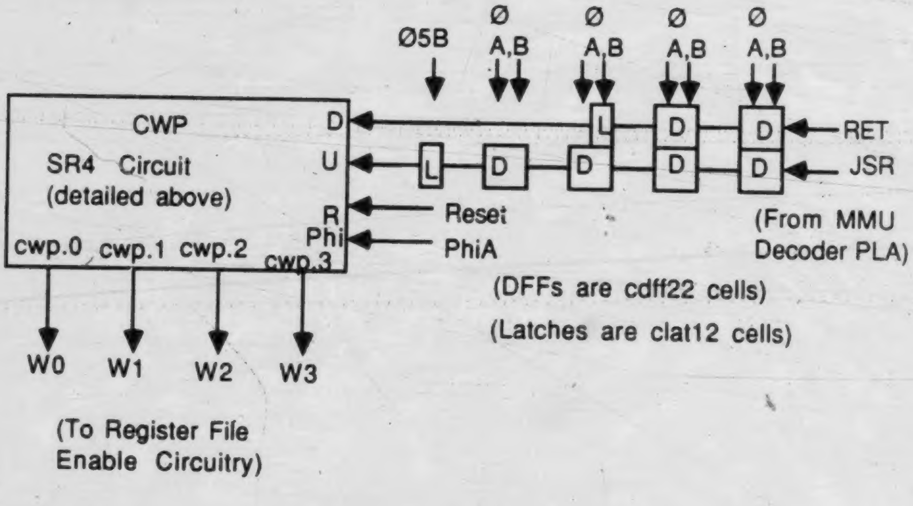

Ø5B
Ø
A,B
Ø
A,B
Ø
A,B
Ø
A,B
CWP
SR4 Circuit
(detailed above)
D
U
R
Phi
cwp.0
cwp.1
cwp.2
cwp.3
L
D
RET
JSR
Reset
PhiA
(From MMU
Decoder PLA)
(DFFs are cdff22 cells)
(Latches are clat12 cells)
W0
W1
W2
W3
(To Register File
Enable Circuitry)

Fig. 5.3.7: Current Window Pointer (CWP) Schematic

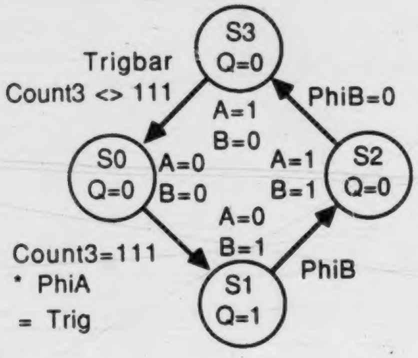
BLANK PAGE

> When a RET instruction is detected by the decoder, the *RET* signal is asserted to the CWP circuitry and latched into a D flip-flop. After an appropriate number of pipestage delays, the SR4 circular shift register receives the signal to shift down.

Fig. 5.3.8(a) details the design details including the state diagram and state equations of the SWP circuitry. Schematics of the SWP.UP and SWP.Down circuitry are given in Fig. 5.3.8(b). Schematics for the 3-bit counter and associated circuitry are given in Fig. 5.3.9. The SWP utilizes a more complex finite state machine and a 3-bit counter to assert the *shift-up* and *shift-down* signals to the shift register. The SWP finite state machine operates as follows:

> When the SWP is given a *reset* signal, bit 0 (W0) of the SR4 is set to logic "1". The other bits, latches and D flip-flops are set to logic "0".
>
> When it becomes necessary to perform a *background* register save, the *Push* and *RegTx* signals are asserted by the off-chip MCU. These signals cause 8 registers (one register subwindow) to be *Push*ed one at a time via the MMU bus and the OUT bus during each subsequent ALU instruction. The 3-bit counter (count3) indicates which register within the subwindow that will be stored on the stack by the next Push operation. Count3 is cleared before the first Push and incremented after each subsequent register push until the 8 registers in the window referenced by the SWP are saved on the register stack of the MCU. The *Trigger* to increment the SWP is asserted when the last register (register 7) of the SWP window is being transferred (count3 = 7) This trigger activates the SWP.UP finite state machine which in turn asserts a *shift-up* signal to the SWP shift register. The SWP.UP finite state machine also disallows further Triggers for 2 clock cycles to prevent glitches.

FSM for Activating SWP.Up and SWP.Down for Exactly 1 Clock Phase


S3
Q=0
Trigbar
Count3 <> 111
PhiB=0
A=1
B=0
S0
Q=0
A=0
B=0
A=1
B=1
S2
Q=0
A=0
B=1
Count3=111
* PhiA
= Trig
S1
Q=1
PhiB

Next State Equations

$A = S1 \cdot PhiB + A \cdot \overline{(S2 \cdot Trigbar)}$

$B = S0 \cdot Trig \cdot PhiA + B \cdot \overline{(S2 \cdot PhiBbar)}$

after simplifying:

$A = A'\,B\,PhiB + A\,B + A \cdot Trig$

$B = A'\,B'\,Trig\,PhiA + A'\,B + B\,PhiB$

$Q = A'B$ or $Q = B$

Fig. 5.3.8 a) Design of SWP Circuitry

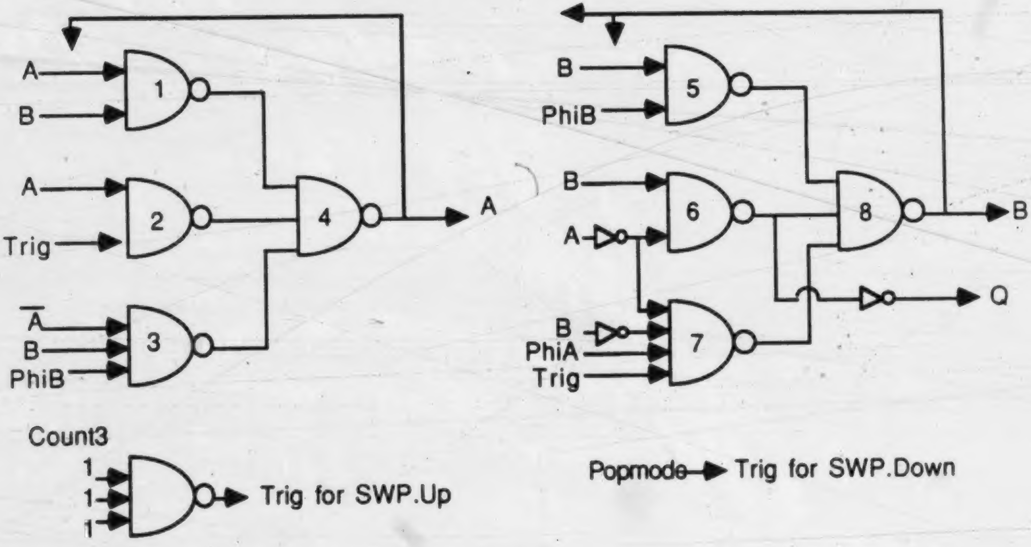

A
B
1
A
Trig
2
A
B
PhiB
3
4
A
Count3
1
1
1
Trig for SWP.Up
B
PhiB
5
B
A
6
B
PhiA
Trig
7
8
B
Q
Popmode
Trig for SWP.Down

Fig. 5.3.8 b): Schematic of SWP Circuitry

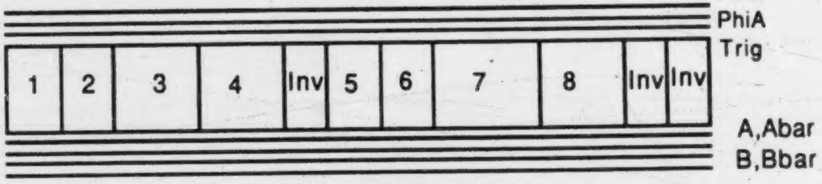

PhiB
PhiA
Trig
1
2
3
4
Inv
5
6
7
8
Inv
Inv
A,Abar
B,Bbar

Fig. 5.3.8 c):
Standard Cell Layout of SWP.Up and SWP.Down FSMs

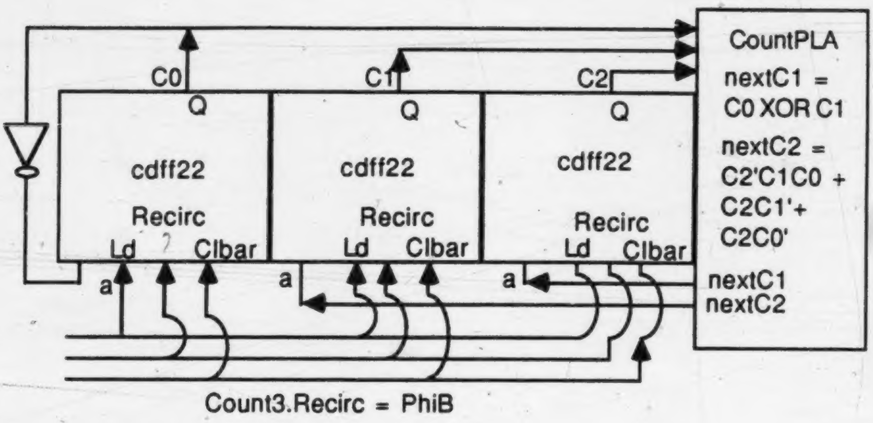

CountPLA
nextC1 = C0 XOR C1
nextC2 = C2'C1C0 + C2C1'+ C2C0'
C0
C1
C2
Q
Q
Q
cdff22
cdff22
cdff22
Recirc
Recirc
Recirc
Ld
Clbar
Ld
Clbar
Ld
Clbar
a
a
a
nextC1
nextC2
Count3.Recirc = PhiB

Fig. 5.3.9 a): 3-bit Counter Schematic

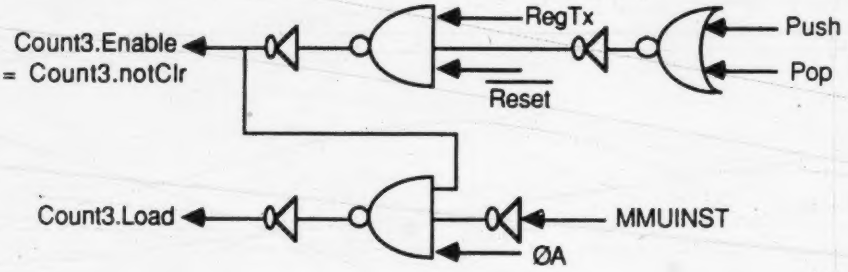

Count3.Enable
= Count3.notClr
RegTx
Push
Pop
Reset
Count3.Load
MMUINST
ØA

Fig. 5.3.9 b): Schematic of 3-bit Counter Input Signals

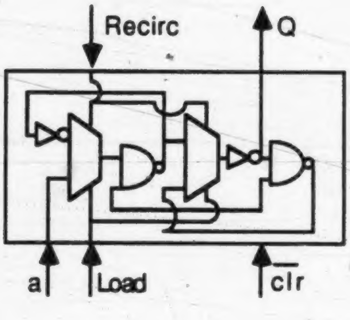

Recirc
Q
a
Load
clr

Fig. 5.3.9 c): cdff22 Standard Cell Schematic

When it becomes necessary to perform a *background* register restore, the *Pop* and *RegTx* signals are asserted. The *Pop* signal is the trigger for the SWP.Down finite state machine so the SWP shift register shifts down one window. The trigger is debounced by two cycles as shown in Fig. 5.3.8(a) but this could be reduced to one cycle to allow SWP to shift down more frequently than once every two cycles. The *RegTx* signal indicates that register transfers are also to occur. The *Pop* and *RegTx* signals cause the 8 registers in the window now referenced by the SWP to be restored from the register stack of the MCU. First the 3-bit counter (count3) is cleared and single registers are *Popd (restored) via the MMU bus and IN bus during each subsequent ALU instruction.*

The schematics for the MMU window and MMU address multiplexing are given in Fig. 5.3.10. The MMU Window multiplexer selects the window of the MMU bus (MW0-3) from the SWP or the CWP. The CWP is selected whenever I/O or Control instructions are being executed and the SWP is selected when ALU instructions are being executed to enable *background* register saves or restores. The MMU address multiplexer selects the address of the register within the MMU window from either an operand address or the output of the 3-bit counter. During *background* register saves/restores the 3-bit counter addresses the register to access the MMU bus. During I/O or control instructions, MMU@ comes from field F3 of the instruction opcode.

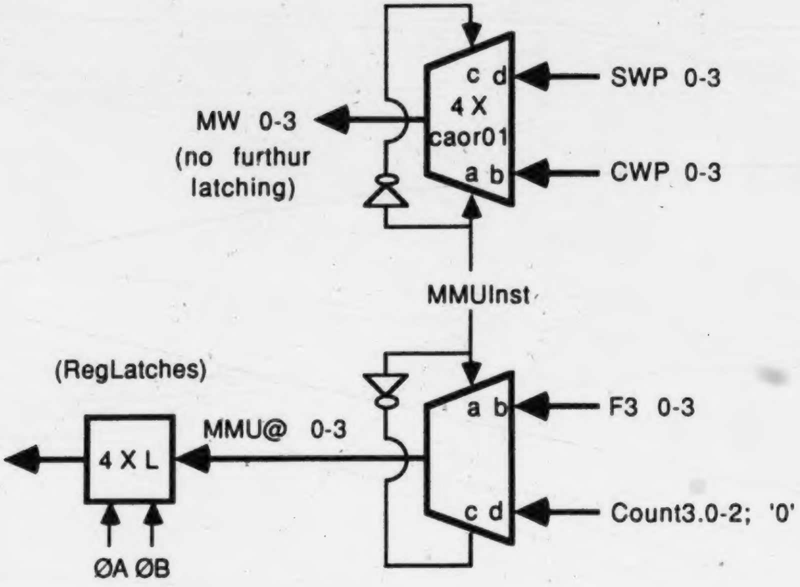

c d
4 X
caor01
a b
SWP 0-3
CWP 0-3
MW 0-3
(no furthur latching)
MMUInst
(RegLatches)
4 X L
MMU@ 0-3
a b
c d
F3 0-3
Count3.0-2; '0'
ØA ØB

Fig. 5.3.10 a): MMU Bus Window and Address Multiplexing

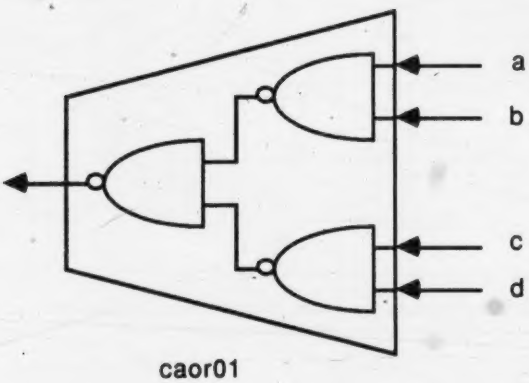

a
b
c
d
caor01

Fig. 5.3.10 b): Schematic for caor01 Standard Cell

6. ANALYSIS OF CONCEPTS

The purpose of this chapter is to analyze and verify the architecture (as presented in Chapter 4) as well as the novel architectural features which were introduced in Chapter 3. The overall architecture is verified by an emulator program as detailed in Section 6.1. Novel features of the architecture are analyzed in Section 6.2 in terms of their advantages and disadvantages.

6.1. Emulation of Architecture

During the Architectural design stage, a high-level emulator was written in the 'C' programming language to define the operation of the ARC processor. A listing of the emulator is provided in Appendix A. It can be used as a reference for the resource scheduling and pipelining details of the instruction set and background operations. The Emulator was tested with several sample programs to exercise the various architectural features of ARC and it verified that there were no resource or timing conflicts.

The emulator was also used to clarify and improve architectural details of the instruction set, the instruction pipeline, resource allocation, and functional details of the IOCU. For example, during the design of the emulator, the typical number of pipestages of transport and control instructions was reduced from 6 to 3. It would have been useful at this stage to have had a suite of common programs with which to test out the architectural features of ARC. This may have lead to additional modifications to the ARC instruction set, particularly to the transport and control instructions, which may have yielded further improvements.

6.2. Analysis of Architectural Features

A cause and effect relationship was presented in Chapter 3 between the novel features of ARC and several advantages. Analysis of these novel features and claimed advantages was performed semi-empirically using a set of ARC code sequences. [1] The code sequences chosen for this purpose include relatively common algorithms including: a) bubble sort, b) multiplication, c) factorial calculation, and d) matrix multiplication. The 'C' and ARC code sequences for each of these algorithms are presented in Appendix B.

---

[1] Given a sequence of instructions for ARC, it is a simple matter to obtain statistical data

From the ARC code sequences, static and dynamic statistics have been obtained. Static statistics treat each of the ARC subroutines as linear code sequences and simply count the number of different instruction types, the size (in words) of the subroutine, etc. Dynamic statistics are obtained by picking a typical set of parameters and calculating how the subroutine would execute with the given parameters. Dynamic statistics can then be obtained including the frequency of instruction types encountered during the example execution of the algorithm. The conditions under which the dynamic statistics were gathered for each of the algorithms are presented in Table 6.1.

Table 6.1: Dynamic Benchmarks for Various Algorithms

Algorithm	Dynamic Benchmark
Bubble Sort	Sorted an array of 100 integers.
Multiplication	A multiplication with the Multiplier set to 127.
Factorial	The calculation of 7!
Matrix Multiplication	The multiplication of a pair of 5 X 5 matrices.

---

such as frequencies of various types of instructions, I/O bandwidth, number of NOPs, etc.

The results of the static and dynamic instruction analysis are presented in Tables 6.2, 6.3, and 6.4. An analysis of this data with respect to each of the design objectives from Chapter 3 is presented in the following subsections.

6.2.1. Bandwidth Utilization/Availability

One goal of the ARC design was to strike a balance between bandwidth utilization and bandwidth availability. A high bandwidth utilization would imply efficient IN and OUT bus resource utilization while a high bandwidth availability means bandwidth available for background register saves/restores as well as LoaD and STore instructions. In this section, the bandwidth utilized by the instruction stream is analyzed during even and odd phases on the IN and OUT busses (refer to Fig. 4.7). Bandwidth requirements of background operations will be discussed in detail in Section 6.2.3.

Instruction fetches use every even cycle on the IN bus. This accounts for 50% of the total IN bus bandwidth and 100% of the even phase IN bus bandwidth. LoaD instructions utilize some of the bandwidth during odd cycles on the IN bus. From Table 6.3, LoaDs account for as much as 11% of the instructions encountered (for Bubble Sorts).[2] An instruction frequency of 11% LoaDs accounts for 11% of the total IN bus bandwidth and 22% of the odd phase IN bus bandwidth.

OUT bus bandwidth is utilized by all transport and control instructions such as LD, ST, BR, JMP, and JSR. The LD, BR, JMP, and JSR instructions utilize one phase on the OUT bus to send out a memory address. The ST instruction requires 2 phases of the OUT bus: one for the memory address, the next for data. Therefore, we can calculate the utilization of the OUT bus according to the following formula:

---

[2] Example subroutines are all processing intensive. It is likely that LoaDs and STores would be more frequent in I/O intensive subroutines. As well, fewer LoaDs and STores occur with this architecture because background register saves and restores ensure that 8 registers of the 16 per window do not need to be saved and restored during subroutine calls and returns.

Table 6.2: Static Instruction Statistics

Algorithm	ALU Instructions	NOPs	Static Push Opportunities	Static Pop Opportunities	Code Size
Bubble Sort	13	7	19	9	18 Words
Multiply	18	1	20	7	15 Words
Factorial	7	5	13	7	8 Words
Matrix Multiply	26	9	25	13	28 Words

Table 6.3: Dynamic Instruction Statistics

(Frequency of instruction types encountered during example execution of Algorithm)

Algorithm	ALU	NOP	LD	ST	SET,SKPONC	BR/JMP	JSR
Bubble Sort	37%	15%	11%	11%	11%	15%	0%
16 bit Multiply	69%	1%	0%	0%	12%	16%	0%
Factorial	28%	39%	0%	0%	8%	16%	8%
Matrix Multiply	52%	17%	6%	3%	4%	6%	12%

Table 6.4: Dynamic Push and Pop Opportunities

Algorithm	Dynamic Push Opportunities	Dynamic Pop Opportunities	Push Opportunities before First JSR	Pop Opportunities after Last JSR
Bubble Sort	49,900	24950	-	-
16 bit Multiply	79	31	-	-
Factorial	61	30	8	4
Matrix Multiply	2100	987	11	9

$$utilization\,(OUT)=2\times freq.\,(ST)+freq.\,(LD)+freq.\,(BR/JSR/JMP)$$

The frequency of the various instructions types can be obtained from Table 6.3. Thus, the OUT bus bandwidth utilized by the ARC instruction sequences (of the example subroutines of Appendix B) accounts for between 16% and 48% of the total OUT bus bandwidth.

According to the above calculations, utilization of the IN bus is approximately 61% and utilization of the OUT bus is 48% or less for the example subroutines. This leaves considerable bandwidth available for background register saves and restores as discussed in Section 6.2.3.

6.2.2. Bandwidth Balancing

The concept of bandwidth balancing was introduced in Chapter 3 as a means of boosting bandwidth utilization and scheduling around pipeline conflicts. An attempt was made to balance the IN and OUT bus even and odd phase bandwidth by scheduling different operations to alternate phases (See Fig. 4.7).

The IN bus is fully utilized during even clock phases for instruction fetches (as discussed in Section 6.2.1). During Odd phases, background register restores (Pops) or data LoaDs may occur. These operations are certain to be much less frequent on average than instruction fetches but, at intermittent times, LoaDs or Pops are equally as frequent. [3] Some pipelining restrictions apply to LoaDs and Pops which serve to slightly reduce the IN bus bandwidth utilization during odd clock phases. In particular, Pops are restricted to occurring during first halfword ALU instructions and LoaDs are restricted to being first-halfword instructions. Therefore, some odd phase IN bus bandwidth can never be utilized (during first halfword ST or control instructions) because Pushes/Pops, transport and control instructions all request service from the MCU via a 4-bit code (Mx0-3). In addition, an internal MMU bus conflict would occur between transport instructions and Pushes/Pops.

---

[3] When entering a procedure, LoaDs are usually frequent as it becomes necessary to load in variables from memory. Pops are frequent when the MCU decides that it is time for ARC to perform background register restores.

The OUT bus is also more highly utilized during even clock phases. Most transport and control instructions (LD, ST, and JSR) send out an address during even clock phases. BR and JMP instructions may send out addresses during either phase. Background register Pushes can occur during either even or odd phases but will occur less often during odd phases because an MMU bus conflict exists between odd phase Pushes and LoaD, STore, and LDSR instructions. STores utilize the OUT bus for addresses during even phases and for data during odd phases. Given the above facts, the OUT bus bandwidth will be more highly utilized during even clock phases by the number of LoaD, and JSR instructions and the slight difference in Pushes between the two phases. From the statistics of Table 6.3, LD and JSR instructions can account for up to 18% of all instructions executed. Therefore, during I/O intensive subroutines or when calls are frequent, considerably more OUT bus bandwidth would be used during even cycles.

It would be possible to better balance the I/O utilization between even and odd clock phases by removing the halfword instruction restrictions of Table 4.2. This would require a more robust and complex pipeline control circuitry. Another improvement would involve removing restrictions on Pushes and Pops. Currently Pushes and Pops cannot occur during control instructions because communication to the MCU is via a 4-bit code (Mx0-3) which notifies the MCU of transport and control requests and Pushes/Pops. An independent Push/Pop notification to the MCU would increase I/O bandwidth utilization, particularly on odd phases of the IN bus.

6.2.3. Bandwidth Available for Background Register Saves/Restores

In the previous 2 sections, it was verified that significant IN and OUT bus bandwidth is available for background register saves and restores to occur. Now it is necessary to analyze any other conditions which influence when background register saves and restores can occur.

To complete a background register save (Push) of a register subwindow of 8 registers, 8 or more ALU instructions not preceded by LD, ST, or LDSR must occur in a subroutine. The minimum number of register pushes is described in absolute terms by the following equation:

$$MinimumPushes = 8 \times (calldepth - registerfiledepth + 2)$$

If insufficient Pushes have been performed, the MCU will have to feed ARC NOPs until sufficient registers have been saved. A register file with 8 windows (register file depth = 8) would enable a code segment with a call/return depth of 6 or less to execute without *necessary* Pushes or Pops (provided that all registers were saved to begin with). In general, more pushes than necessary will be performed to allow for successive calls. The nominal number of pushes may be close to the maximum number of pushes described by the following equation:

$$MaximumPushes = 8 \times (NumberofCalls)$$

A background register restore (Pop) of a register subwindow can occur if 8 or more ALU instructions occupy the first halfword instruction within a subroutine. The minimum number of register Pops is described in absolute terms by the following equation:

$$MinimumPops = 8 \times (returndepth - registerfiledepth + 2)$$

If a return is encountered and an insufficient number of Pops have occurred, it is necessary for the MCU to feed ARC NOPs until sufficient registers have been restored from the register stack. The nominal number of Pops are likely to be close to the minimum because Pops do not have to occur unless the corresponding registers have been overwritten (more than reg_file_depth - 2 successive calls followed by successive returns).

Static push and pop opportunities are presented in Table 6.2 for each of the example algorithms. These statistics assume the code is executed linearly so they can be considered as crude nominal statistics for push and pop opportunities. Dynamic statistics for the iterative example programs are presented in Table 6.4. These statistics are somewhat more useful. The total number of dynamic push and pop opportunities per subroutine are presented in the first 2 columns of Table 6.4. These represent the total amount of bandwidth available for Background register saves and restores. Note that considerably more bandwidth than necessary is available for Pushes and Pops and that available push bandwidth is approximately double the available pop bandwidth (because of halfword restriction).

The number of push opportunities before the first JSR and the number of pop opportunities after the last JSR are given in the third and forth columns of Table 6.4. These statistics are provided because pushes are likely to be required after entering a subroutine but before the first JSR. Similarly, Pops are likely to occur before returning to the calling subroutine. These statistics are also useful for analyzing a worst case scenario — recursive calls or a series of calls/returns much greater than the depth of the register file. The number of these push and pop opportunities seems to be just barely adequate for Pushes but the number of Pop opportunities may be inadequate for recursive routines.

From the discussion above, a number of characteristics of the background register save/restore functions may be noted. Background Pushes and Pops seem to be feasible provided that the considerable amount of available bandwidth is utilized effectively. Advantages include utilizing bandwidth that would otherwise not be used and less instruction overhead during subroutine calls and returns. A disadvantages is the inefficient use of the available bandwidth. Pushes are likely to be more frequent than

Pops and this corresponds with more Push opportunities than Pop opportunities. The algorithm within the MCU to determine when Pushes and Pops occur will have to take into account the above characteristics. An algorithm will have to be selected which effectively utilizes the available bandwidth and allows for more lead time when directing register Pops.

6.2.4. Nonprocessing States

From Table 6.3, the frequency of NOPs executed is approximately 18% of all instructions (of the sample programs of Appendix B). NOPs are necessary because of the pipeline restrictions placed on transport and control instructions, because 2 ARC instructions are packed per word, and because labels have to be aligned to word boundaries. The frequency of NOPs could be slightly reduced by balancing the halfword alignment of transport and control instructions (Currently, the majority of transport and control instructions are restricted to the first halfword). Alternately, these restrictions could be removed entirely, but this would require more complex pipelining and resource scheduling circuitry.

6.2.5. Two Instructions Per Word

All ARC instructions, with the exception of LDI, are 16 bits in length which results in reduced code size (see Table 6.2) and a reduction in the required IN bus bandwidth. This has allowed ARC to fetch an instruction pair every second cycle while executing one instruction per cycle which results in high instruction throughput. This scheme leaves the remaining half of the IN bus bandwidth available for LoaDed data and background register Pops.

Disadvantages to 2 instructions per word include a complex pipeline, pipeline restrictions, and a limitation in register file addressability.

ARC's instruction pipeline is more complex because instructions are split into first- and second-halfword instructions and because clock cycles are split into even and odd phases. To reduce the complexity of the pipeline, pipeline restrictions are introduced (see Fig. 4.7). Pipeline restrictions lead to NOPs and some wasted bandwidth as discussed in Section 6.2.4.

The main problem with 16 bit instructions is that a limited number of registers may be addressed at one time. Restricting the instructions to 16 bits results in restricting the register addresses to 4 bits. Therefore, only 16 *local* registers of the register file may be accessed in any context. This may, in turn, lead to more memory accesses than necessary if more registers were addressable. Perhaps one answer to this potential problem lies in allowing some of the instructions to be 32 bits in size and to allow some instructions to access global registers. As well, less strict scheduling on the IN bus would allow the instruction stream more than half of the IN bus bandwidth. (In Section 6.2.1, the total IN bus bandwidth utilized due to instruction execution of the sample programs was calculated to be 61%)

6.2.6. External Program Counter

The advantages of ARC's external PC feature are discussed in Sections 3.5 and 3.7. Section 3.5 details the bandwidth advantages and feasibility of an external PC while Section 3.7 deals with philosophical issues.

A reduction in required OUT bus bandwidth is unquestionably an advantage of the external PC feature. With an on-chip PC, assuming 2 instructions per word and throughput of 1 instruction per cycle, 50% of the total output bandwidth would be required for instruction addresses. With the external PC, the only instruction addresses that originate from ARC are generated by JSR and JMP instructions. From Table 6.3, the average frequency of these instructions is less than 20%, corresponding to less than 20% of the total output bandwidth. This reduction in required OUT bus bandwidth leads to almost equal input and output bandwidth requirements (see Section 6.2.1 and 6.2.2). Therefore, unidirectional IN and OUT busses are more effective than address and data/instruction busses of more conventional processors.[4] See Section 6.2.7 for a discussion of the advantages and disadvantages of unidirectional IN and OUT busses.

6.2.7. Unidirectional IN and OUT Busses

ARC incorporated unidirectional IN and OUT busses to achieve the following advantages:

a) more input bandwidth without additional pins,

b) eliminate conflicts between incoming and outgoing data,

c) eliminate bus turnaround delays

d) facilitate pipelined memory access.

A disadvantage is that unidirectional busses do not interface directly with a standard memory subsystem. Example system configurations were presented in Section 4.2.

6.3. Testing Results

Unfortunately, complete testing of the ARC chip was not possible because of a number of layout errors as well as some design errors. One of the most serious errors was the layout of the register file bitslice circuitry. This prevented the testing of most of the ARC instructions because ARC is a load-and-store machine and therefore most instructions access registers.

The correct aspect ratios of the transistors in the register bitslice circuit are as given in Fig. 5.2.3. In the actual layout, the n-type transistors of the crosscoupled inverters are not wide enough to discharge the precharged busses. The result of this error is that registers turn to 1's when they are to conditionally discharge a precharged bus.

---

[4] An external PC could also work with a processor with conventional address and data busses by tristating the address bus of the processor during instruction fetches.

Portions of ARC which were testable and found correct included portions of the IOCU input multiplexing circuitry and decoder. The relative branch instruction was also testable and found to be functionally correct.

7. CONCLUSIONS

The design and implementation of ARC has yielded a number of interesting contributions to research. The focus of this architectural design is to alleviate the processor to memory bandwidth bottleneck which is still a problem with current Von-Neuman architecture processors. A number of novel architectural features were introduced with this design and the advantages of these features were verified in Section 6.2. This 32 bit RISC design was implemented in Northern Telecom's 3 micron CMOS process thus verifying that the architecture could fit onto a die of a very conservative technology. This design was a first attempt at such a project by researchers at this university.

7.1. Architectural Concepts

The architectural features were introduced in Chapters 3 and 4 and analyzed in Section 6.2. These features were found to be basically sound but implementing these features efficiently is complex and more research is needed in many cases. The analysis of Section 6.2 indicates that the current ARC implementation of the novel architectural features is not optimum.

7.1.1. 2 Instructions Per Word

Advantages of 2 instructions per word include less required input bandwidth and a high instruction throughput. Disadvantages include a more complex pipeline and limited register addressability.

In the version of ARC implemented, instruction restrictions were introduced to allow the simplification of the pipeline. This resulted in a large number of NOPs in the sample programs. This tradeoff should therefore be reversed — fewer instruction restrictions and more complex on-chip pipeline hardware for better performance.

The limited register file addressability of 16 bit instructions is a more serious drawback. One answer to this problem may be to allow some instructions to occupy 32 bits and to allow some instructions to access global registers.

7.1.2. External Program Counter

This feature is a solution for the output bandwidth bottleneck of previous RISC designs. For example, in the RISC II, instruction addresses interfere with stored data [Kate83]. This feature enables great output bandwidth saving because the instruction address does not need to be sent from the processor to fetch each instruction word. Because of the saving of output bandwidth, OUT bus bandwidth is available for background processes such as register saves. By combining an external PC with unidirectional IN and OUT busses, the input bandwidth bottleneck is also alleviated. Advantages of unidirectional IN and OUT busses are presented in Section 6.2.7. The external PC feature of ARC seems to have few disadvantages other than being unconventional. [5]

7.1.3. Unidirectional IN and OUT Busses

The unidirectional bus feature of ARC is perhaps the most radical of the novel features. Because of this feature, ARC's interface to the MCU and memory subsystem is unconventional but versatile nontheless. Substantial advantages of the unidirectional bus feature of ARC are presented in Chapter 3 and Section 6.2.7 which outweigh its non-conformity.

---

[5] An external PC only works with simple addressing modes such as in a load-and-store RISC architecture.

7.1.4. External Memory Control Unit

To extend the architectural features of ARC to the system level, an external Memory Control Unit (MCU) is required. It operates as a coprocessor which handles the memory interface, instruction fetches, interrupts, background register saves/restores and various other duties. The MCU facilitates several philosophical goals such as locality of information processing and parallelism.

7.1.5. Background Register Saves/Restores

The intent of background register saves and restores is to take away some of the LoaD and STore overhead which normally occurs during procedure calls and returns. The analysis of Section 6.2.3 indicated that the current implementation would probably perform adequately but there is considerable room for improvement (larger register file, fewer conflicts with instructions). The analysis also implied that background registers saves/restores would leave considerable bandwidth available for other background operations such as a data cache. [6]

7.2. Further Work

A number of improvements to the instruction set, architecture, and hardware of ARC are suggested as follows.

7.2.1. Enhancements to Instruction Set

For the most part, the selection of ARC's instructions was good enough to perform most tasks elegantly. However, a number of shortcomings became noticeable during the analysis of the design. Improvements to various instructions are suggested as follows.

---

[6] It is to be noted that background register save/restore circuitry requires a relatively small amount of hardware/area. This is one advantage over an on-chip cache.

Externally decoding the relative BRanch instruction in the MCU would result in a shorter pipeline for this instruction. Additionally, the BRanch instruction would not be delayed (instructions immediately following branch always executed) as is currently the case. This enhancement is introduced in Section 4.2.1.1.

A number of improvements should be made to the SET instruction:

a) The SET instruction should be able to test 16 conditions (as in the MIPS) instead of just the 8 conditions listed in Table 4.1. This is suggested because some of the branch conditions of the example programs in Appendix B were somewhat clumsy to implement. A performance improvement should result with only small additions to Hardware.

b) The SET instruction is not elegant because the S1 and DST addresses share the same operand field. This situation could be slightly improved if, instead, the S2 and DST addresses shared the same operand field (unary tests such as =0, ODD, OV would not overwrite tested register). For tests which involve 3 registers, the destination register could be interpreted as register S2+1 (Difficult to implement and not orthogonal). Alternately, a 4-bit condition mask or a condition flag could be provided but this alternative would require additional instructions to access the condition mask or flag.

The SKPONC instruction could easily become a conditional branch if the F2 and F3 fields are used for a relative branch address.

In order to access 16 or more special registers of the MCU, the LDSR and STSR instructions could use the F2/F3 field(s) to indicate the external register. This instruction could be decoded before sending to ARC or the number of the requested register could be sent via the OUT bus to the MCU.

7.2.2. Improvements to Architecture/System

A number of improvements can be made to the architecture of the ARC as well as to the architecture of the ARC system. Improvements to the ARC chip could include alterations to pipelining, resource scheduling, bandwidth utilization, and exception handling. ARC's system interface is open to further design possibilities because the MCU has yet to be designed.

Pipelining and resource scheduling improvements could include the elimination of halfword instruction restrictions or a better balance between transport and control instructions. This was discussed in Sections 6.2.4 and 6.2.5. Another pipelining improvement would be to prefetch registers during the decode phase so that the register fetch is not during the critical path. This is discussed in more detail in the next section.

Exceptions and interrupts are handled by the MCU. For this to work, all special registers and flags must be accessible to the MCU. Therefore the carry and overflow flags should be implemented off-chip. If all special registers and flags are off-chip, they do not need to be saved when an exception occurs. Thus ARC can perform a context switch with merely a JSR instruction. The handling of interrupts/exceptions is presented in more detail in Sections 3.6, 3.7, and 4.2.

To verify the advantages of the above proposed alterations to ARC, the ARC emulator could be used in conjunction with a compiler and a suite of benchmark programs to obtain statistical performance results. For example, this technique can be used to investigate different instructions, compiler restrictions, and pipeline scheduling. 7

To analyze the effectiveness of the ARC system, an emulator should be written for the MCU and memory subsystem. The system emulator could be used to analyze

---

[7] Most analyses would require minor changes to the emulator.

various IN and OUT bus interfaces to the MCU and the memory subsystem. As well, it would be worthwhile to investigate a means of interfacing static RAM caches to the MCU or IN and OUT busses.

7.2.3. Improvements to Hardware

In addition to the architectural improvements suggested above, a number of modifications to the full custom layout of ARC are necessary to get it working. Hardware redesign is necessary for the following Circuitry:

- The register file bitslice circuitry (Fig. 5.2.3)
- The Control Operations D_w circuitry (Fig. 5.3.5)
- mmout and mmin signals of the IOCU
- IN12 and IN13 shorted together in IOCU

The redesign of the register file bitslice circuitry is a nontrivial endeavor because the area required by the register file is critical. The pitch of the register file bitslice sets the pitch of the ALU and IOCU bitslice circuitry.

The D_w circuitry is composed of standard cells. It could be easily redesigned to match the schematic of Fig. 5.3.5.

The mmout and mmin control signals of the IOCU need to be delayed by an additional clock phase to match ARC's pipeline.

During testing it was discovered that IN12 and IN13 were shorted together. It is possible that there exist other layout errors such as this because ARC was predominantly a full-custom design. It is strongly recommended that future designs utilize software packages such as automatic place-and-route, extraction, and functional simulation.

In addition to the above modifications, the following two enhancements to the first design are suggested.

Registers could be prefetched during the decode stage in latches adjacent to the ALU and IOCU. This would take register access out of the execution pipestage.

Better communication between ARC and the MCU is also desirable. In particular, the Push/Pop handshaking should be improved to make it more versatile and to make it independent of transport and control instructions. Better synchronization of instructions such as BR and LDSR/STSR is also desirable.

REFERENCES

[Bere87]
A. D. Berenbaum et al., "CRISP: A Pipelined 32-bit Microprocessor with 13-kbit of Cache Memory" IEEE Journal of Solid State Circuits, Vol sc-22, No. 5, pp. 776-782, October, 1987.

[Fors87]
M. Forsyth et al., "A 32-bit VLSI CPU with 15-MIPS Peak Performance," IEEE Journal of Solid-State Circuits, Vol. sc-22, No. 5, pp. 768-775, October 1987.

[Furl89]
B. Furlow, "Caching Catches a Ride on 32-bit Designs", ESD: The Electronic System Design Magazine, pp. 36-44, May 1989.

[Gima87]
C. E. Gimarc, "A Survey of RISC Processors and Computers of the Mid-1980s," IEEE Computer pp. 59-69 September, 1987.

[Henn82]
J. L. Hennessy et al, "Hardware/Software Tradeoffs for Increased Performance," Proceedings of the ACM Symposium on Architectural Support for Programming Languages and Operating Systems, Palo Alto, Calif., March 1982, pp. 2-11.

[Henn83]
J. L. Hennesy "PostPass Code Optimization of Pipeline Constraints," ACM Transactions on Programming Languages and Systems, Vol. 5, No. 3, pp. 422-448, July 1983.

[Horo87]
M. Horowitz et al., "MIPS-X: a 20-MIPS Peak, 32-bit Microprocessor with On-Chip Cache," IEEE Journal of Solid State Circuits, Vol sc-22, No. 5, pp. 790-799, October, 1987.

[Hunt87]
C. B. Hunter, "Introduction to the Clipper Architecture," IEEE Micro pp. 6-26 August, 1987.

[John87]
M. Johnson, "System Considerations in the Design of the AM29000," IEEE Micro pp. 28-41 August, 1987.

[Kado87]
H. Kadota et al., "A 32-bit CMOS Microprocessor with On-Chip Cache and TLB," IEEE Journal of Solid State Circuits, Vol sc-22, No. 5, pp. 800-807, October, 1987.

[Kate83]
M. Katevennis, "Reduced Instruction Set Architectures for VLSI," Ph.D. dissertation, Computer Sci. Dep., Univ of California at Berkeley, Oct. 1983. Reprinted by MIT Press, Cambridge, MA, 1985.

[Ly87]
Ty An Ly, "VLSI Design of a CMOS ALU", EE601 Student Report, January 22, 1987. Department of Electrical Engineering, University of Alberta.

[Mitc87]
Alan Mitchell, "Design Optimization of the ARC Carry Select Adder" EE653 Student Report, December 9, 1987

[Neff86]
L. Neff, "Clipper Microprocessor Architecture Overview" Proceedings of Compcon, March, 1986. pp 191-195

[Patt80]
D.A. Patterson and D.R. Ditzel "The Case for the Reduced Instruction Set Computer." Computer Architecture News 9,3 October 1980.

[Patt82]
D. Patterson and C. Sequin "A VLSI RISC," Computer, pp. 8-22 September, 1982.

[Przy84]
S. A. Przybylski et al., "Organization and VLSI Implementation of MIPS," Technical Report No. 84-259, Stanford University, Stanford, Calif., April 1984.

[Raga83]
R. Ragan-Kelly, "Applying RISC Theory to a Large Computer," (Pyramid 90X), Computer Design pp 191-196, Nov 1983.

[Stal88]
W. Stallings, "Reduced Instruction Set Computer Architecture," Proceedings of the IEEE Vol 76 No 1, pp. 38-55, Jan 1988.

[Taba87]
D. Tabak, "Reduced Instruction Set Computer -RISC- Architecture", ISBN 0 86380 047 5, Research Studies Press Ltd., Letchworth, England, 1987.

[TaSe83]
Y. Tamir, C.H. Sequin, "Strategies for Managing the Register File in RISC", IEEE Trans. on Computers, Vol. C-29, No. 2, pp. 44-51.

[West85],
N. Weste and K. Eshraghian, "Principles of CMOS VLSI Design", Addison Wesley Publishing Co ISBN 0-201-08222-5.

APPENDIX A: ARC Emulator Listings

```
/*
 * File: arcsim.c
 *
 * Purpose: main(), unix interface, instruction stack
 *           and other functions of ARC emulator
 */

#include <stdio.h>
#include "opcodes.h"

#define PIPELEN 7
#define NUMFIELDS 4
#define HIGH 0xFFFFFFFF
#define NOP 0x20002000          /* nop is : AQ 0 0 0; AQ 0 0 0 */

/* External Global variables */
extern int s1bus, s2bus, dstbus, mmubus, inbus, outbus;
extern int s1addr, s2addr, dstaddr, mmuaddr;
extern int mmuop;
extern int CWP;
extern int Push, Pop, Regtx;

/*
 * The instruction pipeline of ARC is approximated by
 * the following instruction stack and skipflags
 * The skipflags indicate if corresponding instruction
 * in pipeline is to be skipped
 */
int istack [NUMFIELDS] [PIPELEN];
int skipflags [PIPELEN];

main(argc,argv)
        int argc, *argv[];
{
FILE *infile;
int phasecount=0;

        switch(argc) {
                case 1: infile = stdin; break;
                case 2: infile = fopen( (*(argv+1)), "r" ); break;
                default: printf("usage: %s [inputfile]0,(**argv) );
        }

        CWP = 0;
        istackinit();
        reginit();
        while ( ! done() ) {
                /* do PhiA stuff first */
                precharge();
                /* now do PhiB stuff */
                read_inbus(infile,phasecount);
                iprint(phasecount);
                iprocess();
                PushPop();
                PushFSM();
                PopFSM(phasecount);
                busprint();
                phasecount ++;
        }
        regprint();
 }

precharge() {
        s1bus = HIGH;
        s2bus = HIGH;
        mmubus = HIGH;

        /* this is a kludge */
        mmuop = NOMMU;
}

read_inbus(infile,absphase)
FILE *infile;
int absphase;
{
static int endinput=0;
int count;
int emcpop, emcpush, emcregtx;
char s[128], *sptr;

        /*
          instruction fetch every even phase
          read inbus every phase from hex input file
          when no input left, input = HIGH
          if line begins with '#', it is a comment
        */

        ipush();/* push 1 instruction each phase */
        if ( ! endinput) {
                sptr = fgets(s,128,infile);

                while ( (sptr != NULL) && (s[0] == '#') )
                        /* read next line until not a comment */
                        sptr = fgets(s,128,infile);

                if (sptr == NULL) endinput = 1;
                else {
                        /* read input in hex format */
                        if ( (count = sscanf(s,"%x %d%d%d",&inbus,
                        &emcpush,&emcpop,&emcregtx)) < 1) {
                                endinput = 1;
                                inbus = HIGH;
                        }
                        if (count < 4) {
                                emcpush =0;
                                emcpop = 0;
                                emcregtx = 0;
                        }
                        Pop = emcpop;
                        Push = emcpush;
                        Regtx = emcregtx;
                }

                if ( (!endinput) && ( ! (absphase & 0x01)))
                        ifetch();
        }
}

istackinit() {
int i,j;
        /*
          initialize instruction stack by filling it with NOPs(or AQ 0 0 0)
          and setting all skipflags to 1
        */
        for (i=0; i < PIPELEN; i++) {
                skipflags[i] = 1; /* set skipflags to 1 before instns start */
                istack [0] [i] = AQ_OP;
                for (j = 1; j < NUMFIELDS; j++)
                        istack [j] [i] = 0;
        }
        skipflags[0] = 0;
        /*
          also set Push, Pop, and Regtx to 0
        */
        Push = 0; Pop = 0; Regtx = 0;
}

ipush () {
/* pushes 1 NOP on the instruction stack */
int i,j;

        for (i=PIPELEN-1; i > 0; i--) {
                for (j=0; j < NUMFIELDS; j++) {
                        istack[j] [i] = istack [j] [i-1];
                        skipflags [i] = skipflags [i-1];
                }
        }
        istack [0][0] =AQ_OP;
        istack [1][0] =0;
        istack [2][0] =0;
        istack [3][0] =0;
        skipflags [0] =1;
}

ifetch () {
/* read 2instructions from standard input and place in istack */

        istack [0][1] = (inbus >> 28) & 0x0F;
        istack [1][1] = (inbus & 0x0F000000) >>24;
        istack [2][1] = (inbus & 0x00F00000) >>20;
        istack [3][1] = (inbus & 0x000F0000) >>16;
        istack [0][0] = (inbus & 0x0000F000) >>12;
        istack [1][0] = (inbus & 0x00000F00) >>8 ;
        istack [2][0] = (inbus & 0x000000F0) >>4;
        istack [3][0] = (inbus & 0x0000000F);
        skipflags[0] = 0;
        skipflags[1] = 0;
}

iprint(phasecount)
int phasecount;
{
int i;
        printf("The Inst. Stack and Busses for Phase #%d:0,phasecount);
       printf("-----------------------------------------0);
        printf("Phi  Inst  Skip0);
        for (i=0; i<PIPELEN; i++)
                printf("%d: %x %x %x %x  %d0,i,(istack[0][i]),(istack[1][i]),
                (istack[2][i]),(istack[3][i]),skipflags[i]);
        printf("0);
        printf("Push=%d Pop=%d Regtx=%d0,Push,Pop,Regtx);
}

busprint() {

        printf("inbus=0x%x outbus=%x0,inbus,outbus);
        printf("s1@=%x s1bus=%x, s2@=%x s2bus=%x0,s1addr,s1bus,s2addr,s2bus);
        printf("mmu@=%x mmubus=%x, dst@=%x dstbus=%x0,
                mmuaddr,mmubus,dstaddr,dstbus);
        printf("mmuop=0x%x0,mmuop);
        printf("________________________________________0);
}

iprocess() {
```

```
int phase;

        for (phase=0; phase < PIPELEN; phase++) {
                if ( ! (skipflags [phase]) )
                        decode(phase);
        }
}

done() {
/*
        For now we know we are done if all skipflags
        for all instructions in the pipeline are set to 1.
*/
int i;
        for (i=0; i<PIPELEN; i++) {
                if ( ! (skipflags [i]) )
                        return(0);
        }
        return(1);
}
```

```
/*
 * File: decoders.c
 *
 * Purpose: decoders for different instruction types of ARC
 *
 */

#include <stdio.h>
#include "opcodes.h"

#define PIPELEN 7
#define NUMFIELDS 4

/* instructions currently in the instruction queue */
extern int istack [NUMFIELDS] [PIPELEN];
extern int skipflags [PIPELEN];

decode(phase)
int phase;
{
int F1, F2, F3;             /* 4-bit instruction Fields */
int s1, s2, dst;  /* instruction operands */
int D6_D0;                  /* ALU Control Signals */

        F1 = istack[1] [phase];         /* usually s2 operand */
        s2 = F1;
        F2 = istack[2] [phase];         /* usually s1 operand */
        s1 = F2;
        F3 = istack[3] [phase];         /* usually Dst operand */
        dst = F3;

        switch( istack [0][phase] ) {
        case ADD_OP:
                D6_D0 = 0;
                aluexec(phase,D6_D0,s2,s1,dst);
                break;
        case AQ_OP:
                D6_D0 = 04;
                aluexec(phase,D6_D0,s2,s1,dst);
                break;
        case SUB_OP:
                D6_D0 = 010;
                aluexec(phase,D6_D0,s2,s1,dst);
                break;
        case SQ_OP:
                D6_D0 = 014;
                aluexec(phase,D6_D0,s2,s1,dst);
                break;
        case AWC_OP:
                D6_D0 = 02;
                aluexec(phase,D6_D0,s2,s1,dst);
                break;
        case SWC_OP:
                D6_D0 = 012;
                aluexec(phase,D6_D0,s2,s1,dst);
                break;
        case AND_OP:
                D6_D0 = 070;
                aluexec(phase,D6_D0,s2,s1,dst);
                break;
        case OR_OP:
                D6_D0 = 072;
                aluexec(phase,D6_D0,s2,s1,dst);
                break;
        case XOR_OP:
            D6_D0 = 074;
            aluexec(phase,D6_D0,s2,s1,dst);
            break;
        case BIN_OP:
            D6_D0 = 066;
            aluexec(phase,D6_D0,s2,s1,dst);
            break;
        case BEX_OP:
            D6_D0 = 064;
            aluexec(phase,D6_D0,s2,s1,dst);
            break;
        case SET_OP:
            D6_D0 = 0;
            decode_set(phase,F1,F2,F3);
            /* operands are rearranged for set instruction
               dst is really F2 Field, s1=dst, s2 is F3 Field
            */
            break;

        default:
            decode_2opinstns(phase);
        }
}

decode_set(phase,cond,s1,s2)
/* s1 = dst for this instruction */
int phase, cond, s1, s2;
{
int dst, D6_D0;

        dst = s1;

        switch(cond) {
        case 2:         /* s1 == s2 */
            D6_D0 = 0111;
            aluexec(phase,D6_D0,s2,s1,dst);
            break;
        case 6:         /* s1 == 0 */
            D6_D0 = 0103;
            aluexec(phase,D6_D0,s2,s1,dst);
            break;
        case 9:         /* s1 < s2 */
            D6_D0 = 0170;
            aluexec(phase,D6_D0,s2,s1,dst);
            break;
        case 1:         /* s1 <= s2 */
        case 3:
            D6_D0 = 0150;
            aluexec(phase,D6_D0,s2,s1,dst);
            break;
        case 13:        /* s1 ODD */
        case 15:
            D6_D0 = 0162;
            aluexec(phase,D6_D0,s2,s1,dst);
            break;
        case 0:         /* s1 LT s2 */
            D6_D0 = 0110;
            aluexec(phase,D6_D0,s2,s1,dst);
            break;
        case 12:        /* OV */
        case 14:
            D6_D0 = 0122;
            aluexec(phase,D6_D0,s2,s1,dst);
            break;
        case 8:         /* s1 != s2 */
```

```
	case 10:
		D6_D0 = 0130;
		aluexec(phase,D6_D0,s2,s1,dst);
		break;
	default:
		fprintf(stderr,"Invalid SET Instruction0);
	}
}

decode_2opinstns(phase)
int phase;
{
int opcode, s1, s2, dst;
int D6_D0;

	opcode = 16 * (istack [0] [phase]) + istack[1] [phase];
	s1 = istack[2] [phase];
	/* s2 is part of opcode */
	s2 = istack[1] [phase];
	dst = istack[3] [phase];

	switch(opcode) {
	case NOT_OP:
		D6_D0 = 076;
		aluexec(phase,D6_D0,s2,s1,dst);
		break;
	case SL_OP:
		D6_D0 = 040;
		aluexec(phase,D6_D0,s2,s1,dst);
		break;
	case SLC_OP:
		D6_D0 = 042;
		aluexec(phase,D6_D0,s2,s1,dst);
		break;
	case SR_OP:
		D6_D0 = 050;
		aluexec(phase,D6_D0,s2,s1,dst);
		break;
	case SRC_OP:
		D6_D0 = 052;
		aluexec(phase,D6_D0,s2,s1,dst);
		break;
	case ROL_OP:
		D6_D0 = 044;
		aluexec(phase,D6_D0,s2,s1,dst);
		break;
	case ROR_OP:
		D6_D0 = 054;
		aluexec(phase,D6_D0,s2,s1,dst);
		break;
	case ROL8_OP:
		D6_D0 = 060;
		aluexec(phase,D6_D0,s2,s1,dst);
		break;


	/* for all Transport and Control operations dst (F3) field is really
	mmu field */
	case LD_OP:
		ld(s1,dst,phase);
		break;
	case LDPC_OP:
		ldpc(dst,phase);
		break;
	case LDSR0_OP:
```

```
                        ldsr(0,dst,phase);
                        break;
                case LDSR1_OP:
                        ldsr(1,dst,phase);
                        break;
                case LDSR2_OP:
                        ldsr(2,dst,phase);
                        break;
                case LDI_OP:
                        if (ldi(dst,phase) ) {
                                skipflags[0] = 1;
                                skipflags[1] = 1;
                        }
                        break;
                case ST_OP:
                        st(s1,dst,phase);
                        break;
                case JMP_OP:
                        jmp(s1,phase);
                        break;
                case JSR_OP:
                        jsr(s1,phase);
                        break;
                case BRA_OP:
                        bra((s1 * 16 + dst),phase);
                        break;
                case RET_OP:
                        rtn(s1,phase);
                        break;
                case SKPONC_OP:
                        if( skponc(s1,phase) )
                                skipflags[(phase - 1)] = 1;
                        break;
                default:
                        printf("Illegal instruction: %d0,opcode);
                }
}
```

```
/*
 * File: alusim.c
 *
 * Purpose: ALU functions
 */

#include <stdio.h>
#include "regdefs.h"
#include "opcodes.h"
extern int s1bus, s2bus, dstbus;
extern int CWP;
extern int mmuop;

/*
 * OV and Carry Flags should really be stored in external
 * MCU along with other special registers so that ARC does
 * not have to save it's state in the event of an interrupt
 */
static int carry = 0;
static int ov = 0;

#ifdef DEBUG
alutest()
{
int t1, t2;
int tmp = 0;

        regfile[CWP] [0] =9;
        regfile[CWP] [1] = 4;

        tmp = alufunc(0,0,1,2,0,0,&t1);
        printf("result of alu is %d0,tmp);
}
#endif

aluexec(phase,D6_D0,s2,s1,dst)
int phase, D6_D0, s2, s1, dst;
{
int SetCond;

if (phase == 2)
        mmuop = NOMMU;

if (phase == 3) {
        /* disable_alu = 0; dst_en = 1; */

        dstbus = alufunc(D6_D0,s1,s2,&SetCond);

        regbusrw(WRITE,dst,CWP,DSTB);
        }
}

alufunc(D6_D0,s1,s2,SetCond)
        int D6_D0, s1, s2;
        int *SetCond;
{
static int result;
int a,b, D5_D0, tmp;
int constant;


        constant = s2;

        /* D6 indicates if it is a SET cond instruction */
        *SetCond = D6_D0 & 0100;
```

```
*SetCond = *SetCond >> 6;

 /* strip off D6 bit from D6_D0 */
 D5_D0 = D6_D0 & 077;

 regbusrw(READ,s1,CWP,S1B);
 a = s1bus;
 regbusrw(READ,s2,CWP,S2B);
 b = s2bus;

 switch(D5_D0) {
 case 0:
 case 1:
 case 020:
 case 021: result = a + b;
         setaddovc(a,b,result,&carry,&ov);
         break;

 case 04:
 case 05:
 case 024:
 case 025: result = (a + constant);
         setaddovc(a,constant,result,&carry,&ov);
         break;

 case 010:
 case 011:
 case 030:
 case 031: result = (a - b);
         setsubovc(a,b,result,&carry,&ov);
         break;

 case 014:
 case 015:
 case 034:
 case 035: result = (a - constant);
         setsubovc(a,constant,result,&carry,&ov);
         break;

 case 02:
 case 03:
 case 022:
 case 023: result = (a + b + carry);
         setaddovc((a+carry),b,result,&carry,&ov);
         break;

 case 012:
 case 013:
 case 032:
 case 033: result = (a - b + carry);
         setsubovc((a-carry),b,result,&carry,&ov);
         break;

 case 070:
 case 071: result = (a & b);
         break;

 case 072:
 case 073: result = (a | b);
         break;

 case 074:
 case 075: result = (a ^ b);
         break;
```

```
        /* BIN */
case 066:
case 067: result = (a & 0xFFFFFFF0) + (b & 0xF);
        break;

        /* BEX */
case 064:
case 065: result = a & 0xF;
        break;

        /* NOT */
case 076:
case 077: result = ~a;
        break;

        /* SHL */
case 040:
case 041:
        result = a << 1;
        break;

        /* SHLC */
case 042:
case 043: result = (( a << 1) &0xFFFFFFFE) + carry;
        break;

        /* SHR */
case 050:
case 051: result = a >> 1;
        break;

        /* SHRC */
case 052:
case 053: result = ( (a >> 1) & 0xEFFFFFFF ) | (carry << 31) ;
        break;

        /* ROL */
case 044:
case 045:
case 046:
case 047:
        tmp = ((unsigned) a >> 31) & 0x01;
        result = ((unsigned) a << 1) + tmp;
        break;

        /* ROR */
case 054:
case 055:
case 056:
case 057:
        tmp = a & 0x01;
        result = (a >> 1) | (tmp << 31);
        break;

        /* ROL8*/
case 060:
case 061:
        tmp = (a >> 24) & 0x0FF;
        result = (a << 8) + tmp;
        break;

}

/* now check to see if there are any test instructions */
switch(D6_D0) {
        /* = */
        case 0111: if (a == b) result = 1; else { result =0;}
                break;

        /* = 0 */
        case 0103:
        case 0113:
        case 0107:
        case 0117: if (result == 0) result =1; else { result = 0;}
                break;

        /* a < b */
        case 0170:
        case 0171:
                if (a < b) result = 1; else { result = 0;}
                break;

        /* a <= b */
        case 0150:
        case 0151:
                if (a <= b) result = 1; else { result = 0; }
                break;

        /* <> */
        case 0130:
        case 0131:
                if (a != b) result = 1; else { result = 0; }
                break;

        /* s1 LT s2 */
        case 0110:
                if ( ((long) a) > ( (long) b) )  result = 1;
                else { result = 0; }
                break;

        case 0122:
        case 0123:
        case 0126:
        case 0127:
        case 0132:
        case 0133:
        case 0136:
        case 0137:
                if (ov) result = 1; else { result = 0; }
                break;
        }

        return(result);
}

/*
* Function: setaddovc
*
* Purpose: to set the overflow and carry flags for
*               addition operations
*/
setaddovc(a,b,result,carry,ov)
        int a, b, result, *carry, *ov;
{
long longint;

        longint = ((unsigned) a ) + ( (unsigned) b);
        if (longint && 0x100000000)
                *carry =1;
        else
```

```
		*carry = 0;
	if (((a > 0) && (b > 0) && (result < 0)) ||
	((a < 0) && (b < 0) && (result > 0)))
		*ov = 1;
	else
		*ov =0;
}

/*
* Function: setsubovc
*
* Purpose: to set the overflow and carry flags for
*		subtraction operations
*/
setsubovc(a,b,result,carry,ov)
	int a, b, result, *carry, *ov;
{
long longint;

	longint = ((unsigned) a ) + ( (unsigned) ( - b));
	if (longint && 0x100000000)
		*carry = 1;
	else
		*carry = 0;
	if (((a > 0) && (b < 0) && (result < 0)) ||
	((a < 0) && (b > 0) && (result > 0)))
		*ov = 1;
	else
		*ov = 0;
}
```

```
/*
 * File: regsim.c
 *
 * Purpose: contains register access routines
 */

#include "regdefs.h"
#include <stdio.h>

int regfile [NUMWINDOW] [NUMREG];

/* External Global Variables */
extern int s1bus, s2bus, dstbus, mmubus;
extern int s1addr, s2addr, dstaddr, mmuaddr;
extern int CWP, SWP;

reginit() {
/* sets all registers to zero */
int i, j;
        for (i=0; i< NUMREG; i++)
        for (j=0; j<NUMWINDOW; j++)
                regfile[j] [i] = 0;
}

regprint() {
int i;
        for (i=0; i< NUMREG; i++)
                printf("reg # %d is %x0,i,(regfile[CWP] [i]) );
}

regrw(rw,reg,win,data)
        int rw,reg,win,data;
{

        switch(rw) {
        case READ: return(regfile[win][reg]);
        case WRITE: {
                regfile[win][reg] = data;
                }
                break;
        default: printf("unknown register command: %d0,rw);
                return(-1);
        }
}

regbusrw(rw,reg,win,bus)
        int rw,reg,win,bus;
{
        switch (bus) {
                case MMUB:
                        mmuaddr = reg;
                        switch (rw) {
                        case READ: if (mmubus == -1)
                                mmubus = regrw(rw,reg,win,DATA);
                                else
                                printf("mmubus has already been discharged0);
                                break;
                        case WRITE: regrw(rw,reg,win,mmubus);
                        }
                        break;

                case S1B:
                        s1addr = reg;
                        switch (rw) {
                        case READ: if (s1bus == -1)
```

```
                                s1bus = regrw(rw,reg,win,DATA);
                                else
                                printf("s1bus has already been discharged0);
                                break;
                        case WRITE:
                                printf("cannot write to register from s1bus0);
                        }
                        break;

                case S2B:
                        s2addr = reg;
                        switch (rw) {
                        case READ: if (s2bus == -1)
                                        s2bus = regrw(rw,reg,win,DATA);
                                else
                                printf("s2bus has already been discharged0);
                                break;
                        case WRITE:
                                printf("cannot write to register from s2bus");
                                break;
                        }
                        break;

                case DSTB:
                        dstaddr = reg;
                        switch (rw) {
                        case READ: printf("cannot read from reg to dstbus");
                                break;
                        case WRITE: regrw(rw,reg,win,dstbus);
                        }
                        break;
                default: printf("bus type %d not defined0,bus);
                }
        return(0);
} /* end of regbusrw */
```

```
/*
 * File: regdefs.h
 *
 * Purpose: definitions for register and bus access
 */

/* register definitions */
#define NUMWINDOW 4
#define NUMREG 16
#define READ 1
#define WRITE 0
#define DATA 0

/* bus switches */
#define MMUB 0
#define S1B 1
#define S2B 2
#define DSTB 3
```

```
/*
 * File: iocusim.c
 *
 * Purpose: emulation of Transport and control instructions as well
 *          as background register saves and restores.
 */

#include <stdio.h>
#include "regdefs.h"
#include "opcodes.h"

#define NUMFIELDS 4
#define PIPELEN 7
extern int istack[NUMFIELDS] [PIPELEN];

int s1bus, s2bus, mmubus, dstbus, inbus, outbus;

/* register decoder addresses */
int s1addr, s2addr, mmuaddr, dstaddr;

/*
 * mmuop:  4 bit signal sent off chip describing what is
 * requested from external memory controller
 */
int mmuop;

/*
   CWP is 4 bit Shift register indicating windows
   mmuwin is set = SWP (pointer to register window
                which is next to be saved [pushed] )
*/
int CWP = 1;
int SWP = 1;

/*
 3 bit up counter used for Pushing/Popping window pointed to by SWP
 This counter counts through the 8 registers in a subwindow.
 (for now only use up counter. assume external MMU returns in same order)
 State of Push_pop FSM is held in these global variables:
*/
int Count3 = 0;
int Push = 0;
int PushMode = 0;
int Pop = 0;
int PopMode = 0;
int Regtx = 0;
int Reset = 0;

/*
 * Push can occur during even (Phi1) or odd (Phi0) Phases.
 * Notification of Push to EMC via "mmuop" occurs on phase preceding
 * actual Push of a register via MMU and OUT busses.
 * If two ALU instructions occur in a word, 2 Pushes can occur.
 *
 * The logic to determine if a push is to occur (assuming that ARC is
 * in Push mode) is as follows:
 * Check Decode Pipestage:
 *      If (JSR | BRA | RET )
 *              Cannot Push in following phase because these instructions
 *              send an address on the OUT bus during following phase.
 *              As well, They send out their own mmuop to the EMC during
 *              decode pipestage
 *              Therefore "mmuop" not set to Push
 *      else If (LD | LDSR | LDPC | ST)
 *              Cannot Push in following TWO phases
 *              (above instns use MMU bus and or OUT bus in next 2 phases)
 *              Therefore "mmuop" not set to Push in this or next phase
 *      else If (ALU instruction)
 *              mmuop = Push
 *              During the following phase a register will be Pushed
 *              on the MMU bus to the OUT bus.
 */
PushFSM()
{
static int PushNotify = 0;

        /* SWP initially points to the window to be saved */
        if (! PushMode)
                return(1);

        if (PushNotify > 0) {
                /*
                 * mmuop should have been asserted during the previous
                 * decode phase (1 phase before actual Push)
                 */
                PushNotify --;
                regbusrw(READ,Count3,SWP,MMUB);
                outbus = mmubus;

                if( Count3 == 7) {
                        ShiftUp(&SWP);
                        PushMode = 0;
                        Count3 = 0;
                } else
                        Count3 ++;
        }

        /* Here is where we start the clock on the Push
         * Check that a LD, ST, or LDSR Transport instruction is not
         * executing in phase 3 because during data phase
         * these Transport instructions utilize
        * MMU bus and either IN (LD) or OUT (ST) bus.
         */
       if ( ALUinst(2) && ! LDSTinst(3) ) {
                PushNotify++;
                mmuop = PUSH;
        }
} /* end of Push function */


/*
 * POP occurs during Phi1 (odd phases) because IN bus is used
 * for instruction fetches during even phases.
 * Pop mmuop can only accur when decoding an ALU instruction
 * because all transport and control instructions use mmuop
 * and MMU bus and possibly IN bus.
 *
 * Notification of Pop to EMC via "mmuop" occurs on TWO phases preceding
 * actual Pop of data from the EMC via IN and MMU busses.
 * (Pop thus occurs under different circumstances than Push)
 */
PopFSM(absPhase)
int absPhase;
{
static int PopNotify = 0;

        if (! PopMode)
                return(1);
        if (PopNotify >= 2)
                PopNotify = 1;
        else if (PopNotify == 1)
        {
                PopNotify --;
                mmubus = inbus;
                regbusrw(WRITE,Count3,SWP,MMUB);

                if( Count3 == 7) {
                        PopMode = 0;
                        Count3 = 0;
                } else
                        Count3 ++;
        }
        /* Here is where we start the clock on the Pop
        * Must be during Odd Phase (so as not to interfere with IF)
        * Must be decoding ALU instruction (otherwise cannot send mmuop)
         */
       if ( ALUinst(2) && ((absPhase & 1) == 1) ) {
                if (PopNotify != 0)
                        fprintf(stderr,"Something Wrong0);
                PopNotify = 2;
                mmuop = POP;
        }
} /* end of Pop function */

PushPop()
{
/*
 * This function determines if ARC is to enter Push mode or
 * Pop mode. In Push mode, ARC will save 8 registers, 1 at every
 * opportunity as defined by the Push() function
 */
/*
   PushPop() is called every timestep.
   It returns immediately if ((Push && RegTx) || Pop ) is False.
   Otherwise:
        it goes into PushMode if (Push && Regtx)
        it goes into PopMode if (Pop && Regtx)
        it decrements SWP if (Pop)
   While in PushMode or PopMode, a register is transferred
   each time there is an ALU instruction in phase 3 of the pipeline.
*/

        if ( ! PushMode && ! PopMode && ! ((Push && Regtx) || Pop) )
                return(0);

        if (PushMode && Pop) {
                fprintf(stderr,"Cannot be in BOTH Push and Pop modes0);
                exit(1);
        }
        if (PopMode && Push) {
                fprintf(stderr,"Cannot be in BOTH Push and Pop modes0);
                exit(1);
        }
        if (Pop && Regtx)
                PopMode = 1;
        if (Pop) /* just decrement SWP */
                ShiftDown(SWP);
        if (Push && Regtx)
                PushMode = 1;

        if (! (PushMode || PopMode) )
                /*  not just jet */
                return(0);
        /* OK, So now we are in either PushMode or PopMode */
       /* it is up to the Push and Pop functions to get us out
```

```
        of it */

}

ALUinst(phase)
int phase;
{
/* test if an ALU instruction is in named phase of istack */

        if (phase > PIPELEN ) {
                fprintf(stderr,"Only %d pipestages0,PIPELEN);
                exit(1);
        }

        if (istack[0][phase] == 0xF )
                /* Transport or Control Instruction */
                return(0);
        else
                /* ALU instruction */
                return(1);
}

LDSTinst(phase)
int phase;
{
/* test if a LD, LDSR or ST instruction is in named phase of istack */

        if (phase > PIPELEN ) {
                fprintf(stderr,"Only %d pipestages0,PIPELEN);
                exit(1);
        }
        if (istack[0][phase] != 0xF )
                return(0);          /* ALU instruction */

        if (istack[1][phase] == 0x01)
                /* LD Instruction */
                return(1);
        if (istack[1][phase] == 0x03)
                /* ST Instruction */
                return(1);
        if ((istack[1][phase] & 0x04) == 0x04)
                /* LDSR Instruction */
                return(1);

        fprintf(stderr,"Unidentified instruction - bye0);
        exit(1);
}

ld (s1reg, mmureg, phase)
int s1reg, mmureg, phase;
{
        /* phase 2 is decode phase mmuop always sent out on decode phase */
        /* s1reg and mmureg are set up during PhiA */

switch (phase) {
        case 2: mmuop = LD;
                break;
        case 3: regbusrw(READ,s1reg,CWP,S1B);
                outbus = s1bus;
                break;
        case 4: mmubus = inbus;
                regbusrw(WRITE,mmureg,CWP,MMUB);
        }
        return(0);
}
```

```
ldpc (mmureg, phase)
int mmureg, phase;
{
/*
 * LDPC is a special case of LDSR (to follow)
 */
switch (phase) {
        case 2: mmuop = LDPC;
                break;
        case 4: mmubus = inbus;
                regbusrw(WRITE,mmureg,CWP,MMUB);
        }
        return(0);
}

ldsr (srNum,mmureg, phase)
int srNum, mmureg, phase;
{
switch (phase) {
        case 2:
                if (srNum == 0) mmuop = LDR0;
                if (srNum == 1) mmuop = LDR1;
                if (srNum == 2) mmuop = LDR2;
                break;
        case 4: mmubus = inbus;
                regbusrw(WRITE,mmureg,CWP,MMUB);
        }
        return(0);
}

ldi (mmureg, phase)
int mmureg, phase;
{
switch(phase) {
        case 2: mmuop = LDI;
                break;
        case 3:
                mmubus = inbus;
                regbusrw(WRITE,mmureg,CWP,MMUB);
                /* set global flag and register for use by IU */
                mmuaddr = mmureg;
                return(1); /* cancel currently decoding instruction */
        }
        return(0);
}

st(s1reg, mmureg, phase)
int s1reg, mmureg, phase;
{
switch(phase) {
        case 2: mmuop = ST;
                break;
        case 3:
                regbusrw(READ,s1reg,CWP,S1B);
                outbus = s1bus;
                break;
        case 4: regbusrw(READ,mmureg,CWP,MMUB);
                outbus = mmubus;
        }
        return(0);
}

jmp(s1reg, phase)
int s1reg, phase;
{
```

```
/* jmp is delayed by three instructions */
switch(phase) {
        case 2: mmuop = JMP;
                break;
        case 3:
                regbusrw(READ,s1reg,CWP,S1B);
                outbus = s1bus;
                break;
        }
        return(0);
}

jsr(s1reg, phase)
int s1reg, phase;
{
switch (phase) {
        case 2: mmuop = JSR;
                break;
        case 3:
                regbusrw(READ,s1reg,CWP,S1B);
                outbus = s1bus;
                break;
        /* jsr is delayed by 3 instructions */
        case 6:
                ShiftUp(&CWP);
        /* Delayed return -- 1 instruction word following it is executed.
         * Taking into account pipeline delay to Execute Pipestage
         * of following instruction
         */
        }
        return(0);
}

bra(offset, phase)
int offset, phase;
{
/* branch is delayed by 3 instructions */
switch (phase) {
        case 2: mmuop = BRA;
                break;
        case 3:
                outbus = offset;
                break;
        }
        return(0);
}

rtn(s1reg,phase)
int s1reg, phase;
{
switch (phase) {
        case 2:
                mmuop = RTN;
                break;
        case 3:
                regbusrw(READ,s1reg,CWP,S1B);
                outbus = s1bus;
                break;
        /* Delayed return -- 1 instruction following it is executed.
         * Taking into account pipeline delay to Execute Pipestage
         * of following instruction
         */
        case 4:
                ShiftDown(&CWP);
                /* CWP has to change on following PhiA
```

```
                 * but this will suffice because in this emulator
                 * pipestages are executed sequentially from 1 - 5
                 */
        }
        return(0);
}

skponc(s1reg, phase)
int s1reg, phase;
{
int lsb;
switch (phase) {
        case 2: mmuop = SKPONC;
                break;
        case 3:
                regbusrw(READ,s1reg,CWP,S1B);
                if( s1bus & 0x01) {
                        /* return 1 if we are to skip next instruction */
                        return(1);
                }
                break;
        }
        return(0);
}

ShiftUp(WindowPtr)
int *WindowPtr;
{
        *WindowPtr = *WindowPtr << 1;
        if(*WindowPtr > 8)
                *WindowPtr = 1;
}

ShiftDown(WindowPtr)
int *WindowPtr;
{
        if (*WindowPtr == 1)
                *WindowPtr = 8;
        else
                *WindowPtr = *WindowPtr >> 1;
}
```

```
/*
 * File: opcodes.h
 *
 * Purpose: defines for ARC opcodes
 */

/*
        3 operand ALU instructions:

        OP S2 S1 DST/cond
*/
#define ADD_OP          0x0
#define AQ_OP   0x2
#define AWC_OP          0x8
#define SUB_OP          0x1
#define SQ_OP   0x3
#define SWC_OP          0x9
#define AND_OP          0x5
#define OR_OP   0xB
#define XOR_OP          0x7
#define BIN_OP          0x6
#define BEX_OP          0xA
#define SET_OP          0x4

/*
        2 operand ALU instructions:

        OP OP S1 DST
*/
#define NOT_OP          0xEC
#define ROR_OP          0xD8
#define ROL_OP          0xC8
#define SR_OP   0xD0
#define SRC_OP          0xD4
#define SL_OP   0xC0
#define SLC_OP          0xC4
#define ROL8_OP         0xE0

/*
        Other instructions:

        MMU_OP S1 MMU
*/

#define LD_OP           0xF1
#define LDPC_OP                 0xF7
#define LDSR0_OP        0xF4
#define LDSR1_OP        0xF5
#define LDSR2_OP        0xF6
#define LDI_OP                  0xFD
#define ST_OP           0xF3
#define JMP_OP                  0xFB
#define JSR_OP                  0xF9
#define BRA_OP                  0xFA
#define RET_OP                  0xF8
#define SKPONC_OP       0xF0


/*
        these are the internal mmu op codes and the op codes sent
        to the external mmu;
        iocntrl:  4 bit signal sent off chip describing what is
        requested from external memory controller
*/
```

```
#define LD      001
#define LDPC    007
#define LDR2    006
#define LDR1    005
#define LDR0    004
#define ST      003
#define JMP     013
#define JSR     011
#define PUSH    014
#define BRA     012
#define RTN     010
#define POP     016
/* functions External Mem controller does nothing with */
#define LDI     015
#define SKPONC  000
#define NOMMU   017
```

APPENDIX B: ARC Implementation of Example Algorithms

```
/* In place Bubblesort algorithm */
BubbleSort(A,asize)
int A[ASIZE], asize;
{
        int i, deepest;
        int moves = 1;
        int temp;

        deepest = asize -1;
        while (moves) {
           moves = 0;
           for (i=0; i< deepest; i++) {
                if (A[i] > A[i+1]) {
                        temp = A[i];
                        A[i] = A[i+1];
                        A[i+1] = temp;
                        moves = i;
                }
           }
           deepest = moves;
        }
}
```

Fig. 6.1 a) Bubblesort algorithm in 'C'

```
# ARC code for Bubblesort algorithm:
#
# Register allocation:
# (R0 and R1 are parameters to Bubblesort function)
# R0:   A
# R1:   asize
# R2:   i
# R3:   deepest
# R4:   moves
# R5:   temp
#
        SUB R2 R2 R2            i =0;
        AQ  1  R2 R4            moves = i +1;
        SQ  1  R1 R3            deepest = asize-1;
        AQ  0  R0 R0            NOP
WHILE:
        AQ 0  R4 R5             temp = moves;
        SET =0 R5               set if temp !=0
        NEG R5 R5
        SKPONC  R5              skip if reg 5 bit 31 == 1
        BR END                  relative branch to END:
        SUB R4 R4 R4            moves = 0;
FOR:
        AQ 0 R2 R5              temp = i;
        AQ 0 R0 R0              NOP
        SET >= R5 R3            (i >= deepest) ?
        SKPONC R5
        BR WHILE                end of for loop ?
        AQ 0 R3 R4              Deepest = moves
        AQ 0 R0 R0              NOP
        ADD R0 R2 R6            R6 = &A[i]
        LD R6 R5                temp = A[i];
        AQ 1 R6 R9              R9 = &A[i+1]
        LD R9 R7                R7 = A[i+1]
        AQ 0 R5 R8              R8 = R5
        SET <= R8 R7            set R8.31 if R8 <= R7
        SKPONC R8               R8.31?
        BR FOR                  exchange not necessary
        AQ 1 R2 R2              i ++;
        AQ 0 R0 R0              NOP
        AQ 0 R0 R0              NOP
        ST R6 R7                A[i] = A[i+1];
        SQ 1 R2 R4              moves = i-1;
        BR FOR
        AQ 0 R0 R0              NOP
        ST R9 R5                A[i+1] = temp;
        AQ 0 R0 R0              NOP
END:
        RET                     Return From Subroutine
        AQ 0 R0 R0              NOP
```

Fig. 6.1 b) Bubblesort Algorithm in ARC Mneumonics

```
Mult16(mplicand, mplier,product)
int mplicand, mplier, *product;
{
/* 16 bit multiply of positive numbers */
int temp;
int neg = 0;

        *product =0;

        if (mplicand < 0) {
                neg = !neg;
                mplicand = 0 - mplicand;
        }
        if (mplier < 0) {
                neg = !neg;
                mplier = 0 - mplier;
        }
        if (mplicand < mplier) {
                temp = mplier;
                mplier = mplicand;
                mplicand = temp;
        }

        while (mplier != 0) {
                if ((mplier & 01) == 01)
                        *product  = (*product) + mplicand;
                mplicand = mplicand << 1;
                mplier = mplier >> 1;
        }
        if (neg)
                *product = 0 - *product;
        return(*product);
}
```

Fig. 6.2 a) Multiply Algorithm in 'C'

```
#
# ARC code for 16-bit multiply function
#
#       Registers:
#       0:      Multiplicand
#       1:      Multiplier
#       2:      product
#       12:     temp reg
#       13:     negative flag (negate result?)
#       14:     temp reg
#       15:     temp reg
#
MULT:
        SUB R13 R13 R13             Neg = 0
        SUB R15 R15 R15             R15 = 0
        SKPONC R0               R0 (Multiplicand) -ve?
        NEG R13 R13             R13 = !R13
        SKPONC R1               Multiplier -ve?
        NEG R13 R13             R13 = !R13
        NOT R0 R14              R14 = !R0
        SKPONC R14
        SUB R15 R0 R0           R0 = 0 - R0;
        NOT R1 R14
        SKPONC R14
        SUB R15 R1 R1           R1 = 0 - R1;
WHILE:
        AQ 0 R1 R14             R14 = R1
        SET !=0 R14             Multiplier == 0?
        SKPONC R14
        BR ENDWHILE
        AQ 1 R15 R15            R15 = 1
        AND R1 R15 R12              R12 = multiplier & 01
        ROR R12 R12             ror 1 to sign bit
        NOT R12 R12             inverse logic
        SKPONC R12
        ADD R0 R2 R2            Product += Multicand
        BR WHILE
        SHL R0 R0               Multiplicand << 1;
        SHR R1 R1               Multiplier >> 1
        NOT R13 R14             R14 = !neg
ENDWHILE:
        SKPONC R14              skip if !neg
        SUB R15 R2 R2           Product = 0 - Product
        RET
        NOP
```

Fig. 6.2 b) 16 Bit Multiply in ARC Mneumonics

```
factorial(num)
int num;
/* returns num! */
{
int i, torial, temp;

        torial =1;
        for (i=2; i<=num; i++)
                torial = Mult16(torial, i, &temp);
        printf("Factorial of %d is %d0,num,torial);
}
```

Fig. 6.3 a) Factorial Algorithm in 'C'

```
#
# ARC code for factorial function
#
#       Registers:
#       0:      number (Parameter)
#       1:      i
#       2:      torial
#       3:      address of Mult16
#       4:      temp reg
#
FACTORIAL:
        SUB R1 R1 R1            i = 0;
        AQ 2 R1 R1              i =2;
        SUB R2 R2 R2            torial = 0;
        AQ 1 R2 R2 R2           torial = 1;
WHILE:
        AQ 0 R1 R4              R4 = i;
        SET <= R4 R0            i <= number?
        SKPONC R4
        BR ENDWHILE
        AQ 0 R8 R2              R8 = torial (setting parameters for Mult)
        AQ 0 R9 R1              R9 = i      "
        JSR MULT16
        NOP NOP NOP
        BR WHILE
        NOP NOP
ENDWHILE:
        RET
        NOP
```

Fig. 6.3 b) Factorial Algorithm in ARC Mneumonics

```
MultiplyMatrix(A,B,C,arows,acols,brows,bcols)
int arows, acols, brows, bcols;
int A[AROWS][ACOLS];
int B[BROWS][BCOLS];
int C[AROWS][BCOLS];
{
int i, j, ai, bi, aj, bj;
int product;

        for (i = 0; i< arows; i++)
        for (j=0; j<bcols; j++) {
          C[i][j] = 0;
          /* loop for calculation of Cij */
          /*
           * During this part:
           * ai =i; aj =0; bi = 0; bj = j;
           */
          for (aj =0; aj < acols; aj++)
                  C[i][j] += Mult16(A[i][aj], B[aj][j], &product);
        }
}
```

Fig. 6.4 a) Matrix Multiplication Algorithm in 'C'

```
# ARC code for Matrix Multiplication of variable
# sized matrices:
# C[arows][bcols] = A[arows][acols] X B[brows][bcols]
#
# Register allocation:
# (R0 thru R6 are parameters to Bubblesort function)
# R0:       A           R8:         Multiplicand Parameter to Mult16
# R1:       B           R9:         Multiplier Parameter
# R2:       C           R10:        Product Parameter
# R3:       arows
# R4:       acols
# R5:       brows       R13         C[i][j]
# R6:       bcols       R14:        aj
# R7:       i           R15:        j
MATMULT:
          SUB R7 R7 R7                        i = 0;
          SUB R15 R15 R15                     j = 0;
          SQ 1 R7 R7          i = -1;
          SQ 1 R15 R15                        j = -1;
FOR1:     AQ 1 R7 R7                          ++i;
          AQ 0 R7 R9
          SET < R9 R3                         i < arows ?
          SKPONC R9
          BR END              if not goto END
          NOP
FOR2:     AQ 1 R15 R15                        ++j;
          AQ 0 R15 R9
          SET < R9 R6                         j < bcols ?
          SKPONC R9
          BR FOR1             if not goto FOR1
          NOP
          AQ 0 R6 R8                          (parameters for MULT16)
          AQ 0 R7 R9
          JSR MULT16                          R10 = R8 * R9
          NOP                                 3 instns in delayed branch
          SUB R13 R13 R13                     R13 = C[i][j] = 0;
          SUB R14 R14 R14                     aj = 0;
FOR3:     AQ 0 R14 R8
          SET < R8 R4                         aj < acols ?
          SKPONC R8
          BR FOR2             if not goto FOR2
          AQ 0 R7 R9                          R9 = i (parameter for MULT)
          AQ 0 R4 R8                          R8 = acols (parameter for MULT)
          JSR Mult16          R10 = R8 * R9
          NOP
          ADD R0 R14 R12                      R12 = A + aj
          ADD R10 R12 R12                     R12 = &A[i][aj];
          LD R12 R12                          R12 = A[i][j];
          AQ 0 R14 R8                         R8 = aj   (parameter for MULT)
          JSR MULT16                          R10 = R8 * R9
          AQ 0 R6 R9                          R9 = bcols "
          ADD R1 R15 R11                      R11 = B + j
          ADD R10 R11 R11                     R11 = &B[aj][j]
          LD R11 R11                          R11 = B[aj][j]
          AQ 0 R11 R8                         parameter for MULT
          JSR MULT16                          R10 = R8 * R9
          AQ 0 R12 R9                         parameter for MULT
          NOP
          NOP
          ADD R10 R13 R13                     R13 = C[i][j]
          ADD R15 R2 R12                      R12 = C + j
          JSR MULT16                          get & C[i][j]
          AQ 0 R7 R9                          parameter for MULT
          AQ 0 R6 R8                          "
          NOP
          BR FOR3
          AQ 1 R14 R14                        aj++;
          ST R10 R13                          save C[i][j]
          NOP
END:      RET
          NOP
```

Fig. 6.4 b) Matrix Multiplication in ARC Mneumonics